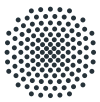


Universität Stuttgart

OSCAR
A textual and spatial exploratory search engine for OpenStreetMap data

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Daniel Bahrdt

aus Stuttgart

Hauptberichter: Prof. Dr. Stefan Funke

Mitberichter: Prof. Dr. Hannah Bast

Tag der mündlichen Prüfung: 6. Dezember 2019

Institut für Formale Methoden der Informatik

2020

Contents

Summary

Searching textual spatial data is a task done everyday for many people. Finding the next supermarket, listing all petrol stations along a journey route or getting an overview of all waterfalls in a given area are just three of many usages of a textual spatial search engine. There are the proprietary search engines from Google, Bing and Yahoo as well as some open source engines like Apache Solr or Nominatim. The later powers the search capabilities of the OpenStreetMap website, a crowd-sourcing effort to create free map data. This dissertation describes OSCAR, an advanced search engine for OpenStreetMap data. It enables a user to combine textual queries with powerful spatial queries while scaling from mobile phones to servers. In our data model we distinguish between regions and items where regions are larger two-dimensional surfaces like countries, cities or suburbs and items are smaller objects like buildings and streets. The line segments of the regions induce an arrangement of cells which is backed by a triangulation. For a cell we have the property that each of its items is covered by the same set of regions. We use these cells as basis for our text search part which has a special emphasis on queries for regions. The combination of the triangulation, an explicit representation of the neighborhood graph of the cells and a hierarchy of the regions allows us to extend the text search part with powerful spatial query operators.

However the geospatial data introduces some problems at various stages of our processing pipeline which we wanted to solve by projecting all points onto the unit sphere. Unfortunately floating-point numbers can only provide trivial solutions to the sphere equation in $\mathbb{R}^2$ and $\mathbb{R}^3$. It is therefore necessary to use rational numbers as an approximation to the real value $x \in \mathbb{R}^3$ obtained from spherical coordinates. We show how to compute an ε-approximation with denominators of at most $10(d-1)/\varepsilon^2$ for any given $\varepsilon \in \left(0, \frac{1}{8}\right]$, improving on a previous result. Based on this method we adapted the CGAL Delaunay triangulation implementation to compute triangulations on the unit sphere $\mathbb{S}^2$.

We conducted extensive benchmarks to show the performance and viability of our approaches on mobile phones and servers. We also compare the effect of our special cell arrangement with more regular spatial subdivisions and find that neither variant dominates with regard to completion timings. We conclude with a comparison of our approach with other state of the art text search engines to show its superiority in the special setting that we examined.

Zusammenfassung

Die Suche in textuell augmentierten räumlichen Daten ist für viele Menschen eine tägliche Angelegenheit. Den nächsten Supermarkt finden, alle Tankstellen entlang einer Reiseroute auflisten oder eine Übersicht über alle Wasserfälle in einer vom Benutzer vorgegebenen Region zu finden sind nur drei Beispiele für die Anwendung von Geo-Text-Suchmaschinen. Neben den großen kommerziellen Suchmaschinen wie Google, Bing und Yahoo existieren auch einige quelloffene Suchmaschinen wie Apache Solr und Nominatim. Letztere bildet die Basis für die Suchfähigkeiten der Webseite des OpenStreetMap Projekts. Diese Dissertation beschreibt OSCAR, eine Suchmaschine für OpenStreetMap-Daten. Sie ermöglicht Nutzern die Suche nach Text mit leistungsfähigen räumlichen Anfragen zu kombinieren. OSCAR kann hierbei sowohl auf einem Mobiltelefon als auch auf einem Server betrieben werden.

Unser Datenmodell unterscheidet zwischen größeren zwei-dimensionalen Regionen und anderen Einzeleinheiten. Beispiele für eine Region sind Deutschland, Stuttgart oder der Stadtteil Stuttgart-Mitte. Innerhalb dieser Regionen existieren die Einzeleinheiten wie z.B. eine Straße oder ein Gebäude. Die Regionen werden durch Polygone dargestellt deren Liniensegmente ein Arrangement von Zellen induzieren welches wir mit Hilfe einer Triangulierung speichern. Für eine Zelle des Arrangement gilt nun, dass jede Einzeleinheit von der gleichen Menge an Regionen überdeckt wird. Auf Basis dieser Zellen entwickeln wir eine Datenstruktur für Textsuchanfragen mit einem Schwerpunkt auf Regionen-Anfragen. In Verbindung mit der Triangulierung, einer explizit Darstellung der Zellnachbarschaft und einer Hierarchie der Regionen ist es uns möglich die Textsuche um leistungsfähige räumliche Operationen zu erweitern.

Leider ergeben sich durch die räumlichen Daten einige Probleme innerhalb unserer Verarbeitungskette. Diese wollen wir lösen, indem wir die Punkte unserer Geodaten auf die Einheitssphäre projezieren und sämtliche Operationen im dreidimensionalen Raum ausführen.

Hierbei hat sich herausgestellt, dass Fließkommazahlen nur triviale Lösungen der Sphärengleichung im drei-dimensionalen darstellen können. Um dennoch die sphärischen Koordinaten unserer Eingabedaten verarbeiten zu können müssen wir rationale Zahlen als Näherung derselben nutzen. Hierfür haben wir eine Methode gefunden mit welcher wir für einen gegebenen Punkt $x \in \mathbb{R}^d$ und einer Genauigkeit $\varepsilon \in \left(0, \frac{1}{8}\right]$ eine Näherung mit Nennern mit einer maximalen Größe von $10(d-1)/\varepsilon^2$ berechnen können. Auf Basis dieser Methode haben wir die Implementierung des

Delaunay-Triangulierungs-Algorithmus des CGAL Projekts erweitert um selbige auf der Einheitssphäre $\mathbb{S}^2$ zu berechnen.

OSCAR besitzt viele Konfigurationsparameter um die Datenstrukturen für einen gegeben Einsatz zu optimieren. In ausgiebigen Benchmarks zeigen wir die Auswirkung dieser Parameter auf die Ausführungszeit und den Platzverbrauch. Diese führen wir sowohl auf einem leistungsstarken Server als auch auf einem älteren Mobiltelefon aus. Ein Vergleich unseres speziellen Zellarrangement mit regulären Raumunterteilungen zeigt, dass keine der untersuchten Varianten in allen Fällen besser ist. Abschließend vergleichen wir unsere Textsuche mit aktuellen Textsuchmaschinen wobei sich herausstellt, dass diese für unsere speziellen Daten signifikant langsamer sind.

1 Introduction

Location-based services are omnipresent in our daily life especially in mobile phones but also widely used on desktop computers. They entail consumer applications like (car) routing, map search and user tracking. There are routing applications not only for cars but also for bicycles, pedestrians, public transportation and many more. These are often combined with a map search facility to locate the current user, provide means to search the destination or simply explore a region of interest. In conjunction with user tracking this allows an integrated system to guide users to places of interest for example to meet your friends or the cab that you just ordered. Geographical Information Systems provide specialized tools for many professions among them road engineering, architecture but also agriculture and ecosystems research. A more playful usage is geocaching – a modern variant of scavenger hunt. Most applications have in common that a powerful map search engine is either a necessity or a helpful addition. The quality of a map search engine heavily depends on the underlying data set. Additionally the type of data also influences the architecture and possible capabilities of the system. One of the largest free geospatial data set is provided by the OpenStreetMap project which saw the day of the light in 2004 and set out to conquer humankind's geographical knowledge in a crowd-sourced mapping effort. As a community project the map quality available from OpenStreetMap heavily depends on its user base. In western countries the data is very detailed containing even single trees. Making this vast data source available to users is the main goal of this dissertation. There are of course other map search engines based on the OpenStreetMap data set among them the projects' official search engine called *Nominatim*. In comparison to the other available systems we strive to provide means to explore the data set hence the subtitle of this thesis. Let us briefly clarify what we mean by this and what the possible benefits are. Traditionally one enters a search query and the system decides which data items are the best match for the query. The user then gets a list of the data items, possibly with a map pin for each entry. There is no indication how large the result is or how it is distributed. Nor are there any hints for possible improvements of the result – a task also called faceted search. See figures 1.1, 1.2 and 1.3 for an example query where we want to locate waterfalls in order to plan a nice hiking trip.

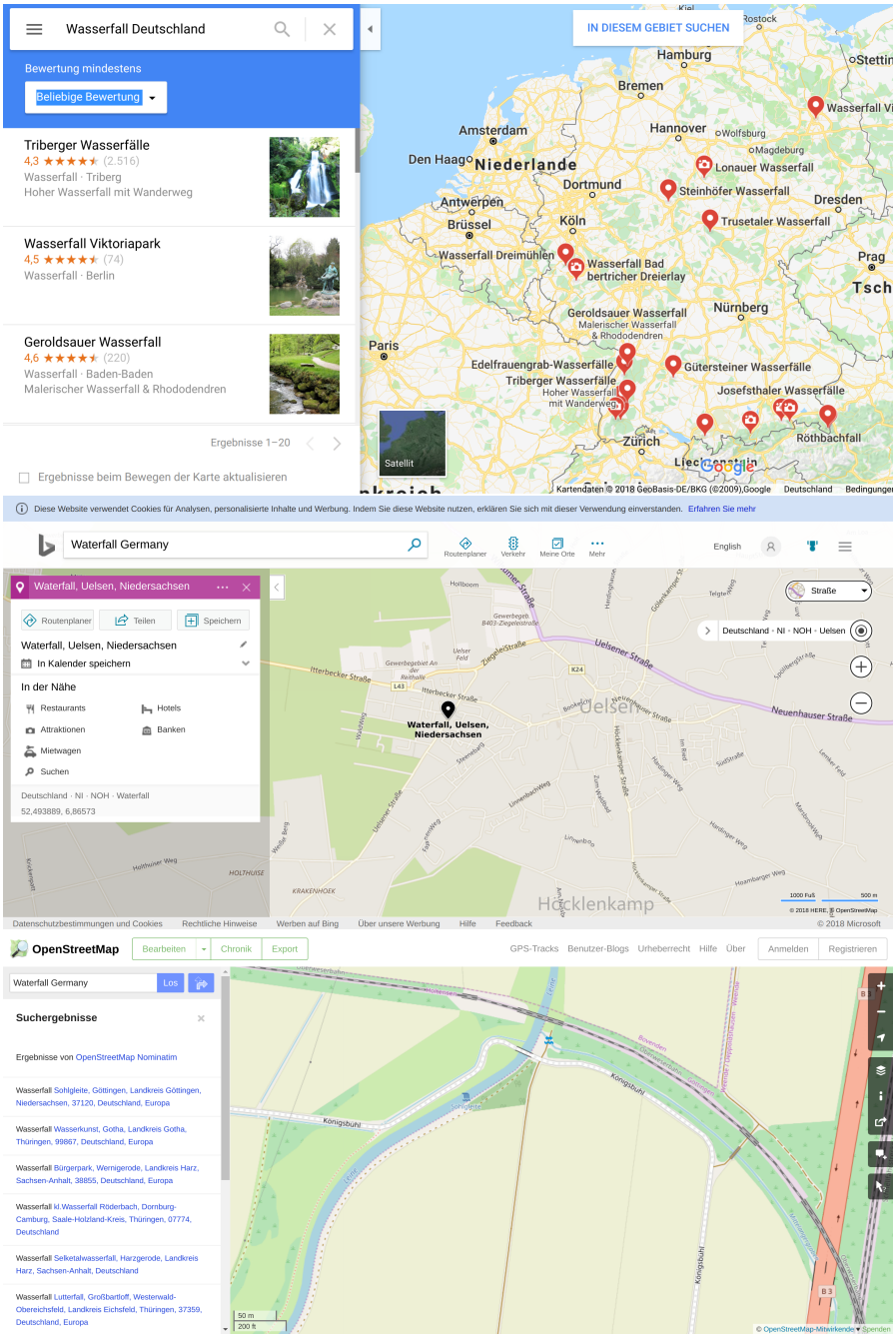


Figure 1.1: Results of trying to find all waterfalls in Germany. Top: Google Maps lists some waterfalls in Germany together with nice pictures. Middle: Bing maps simply zooms to a place named “Wasserfall”. The same happens if we try to get an overview of the restaurants in Germany. Bottom: Nominatim also zooms to a single waterfall but also has a list of other possible candidates. However the list is rather small and it is not possible to visualize the result on the map.

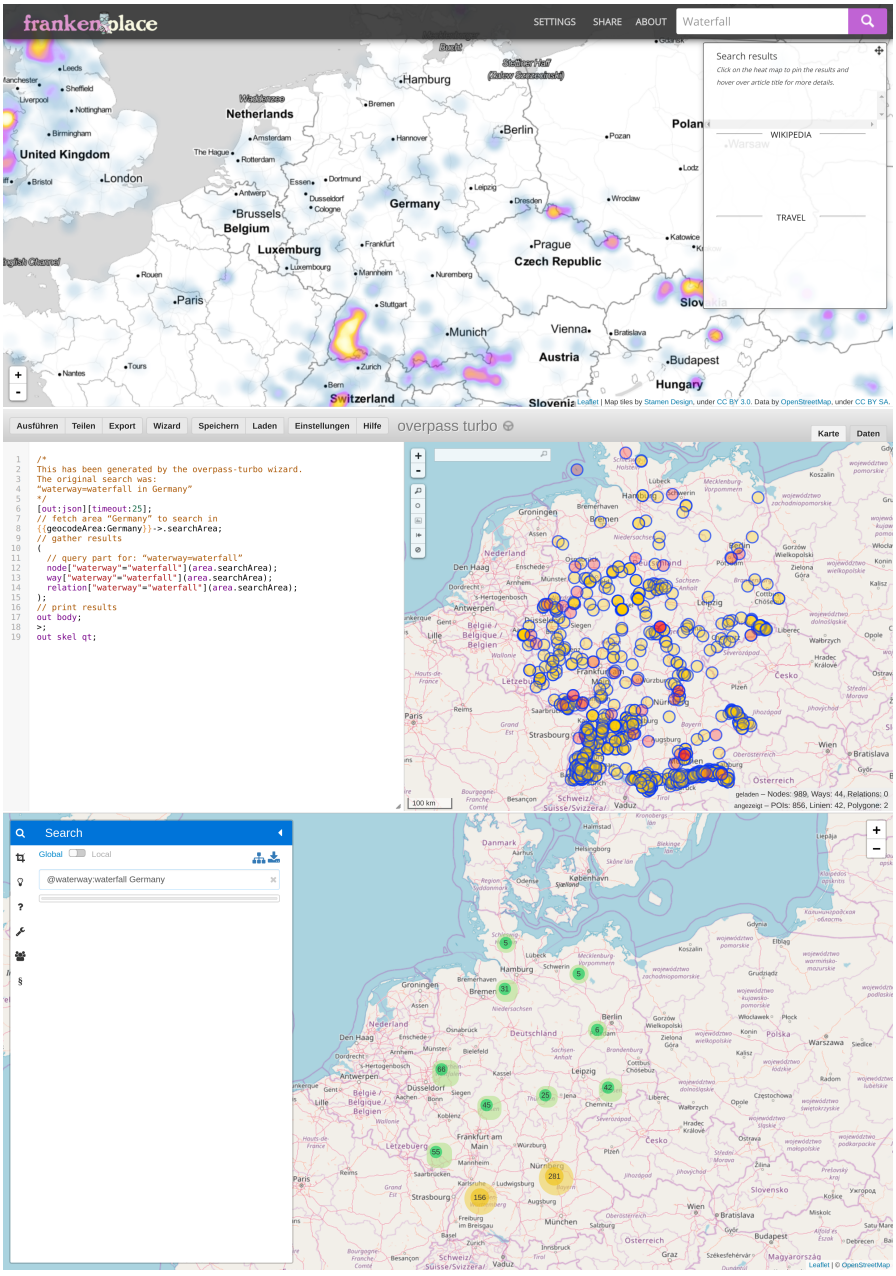


Figure 1.2: Results of trying to find all waterfalls in Germany. Top: Frankenplace, being an exploratory search engine, has a nice visualization using a heatmap which gives a user a good overview in an instant. However no result list is available. Middle: Overpass-Turbo has the ability to list and display all waterfalls. The query is rather slow and the result is simply dumped onto the map. OSCAR: Our system shows clusters with approximate item counts. The result list is displayed on a lower zoom level. Faceted search is provided by [77].

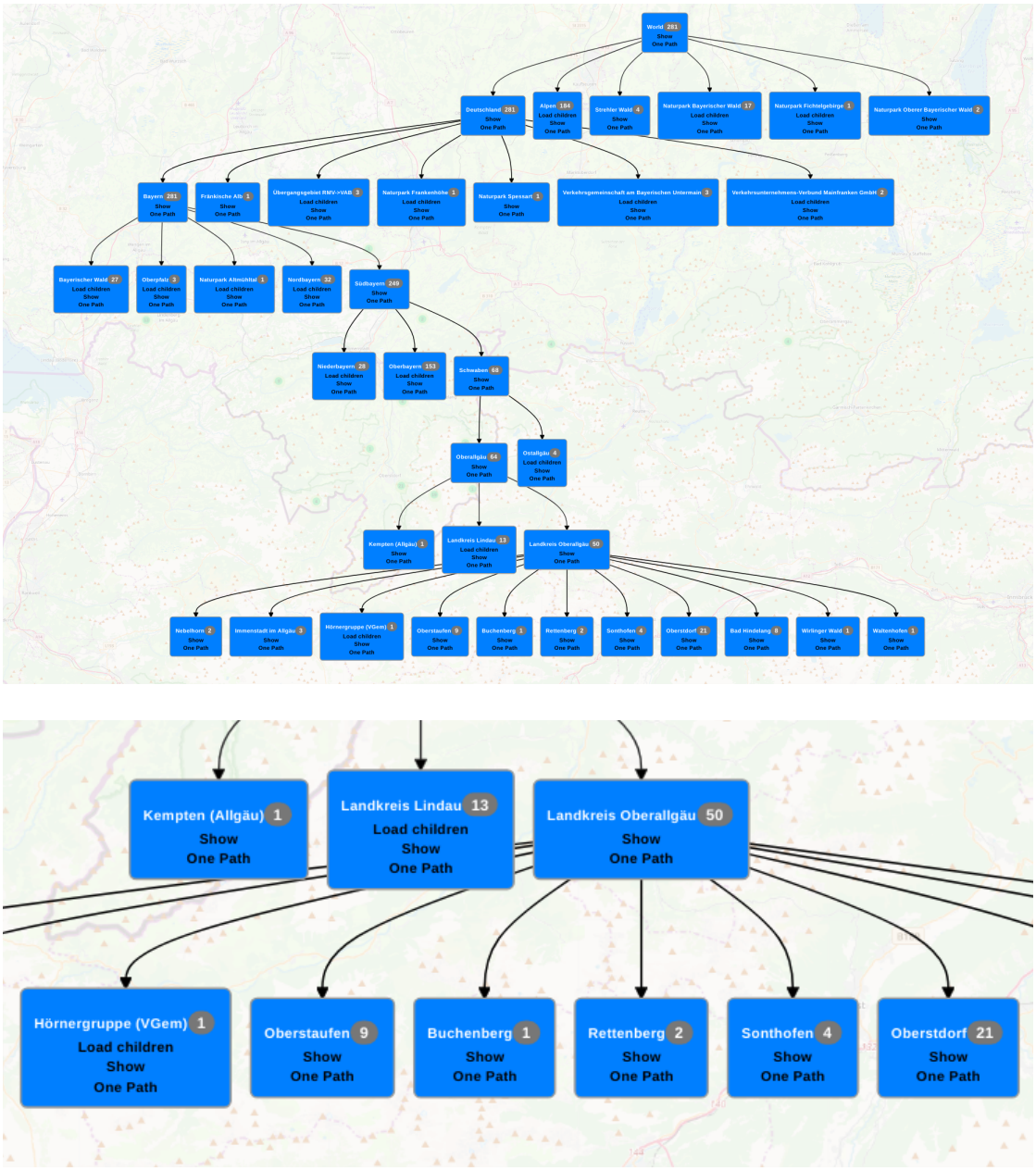


Figure 1.3: Our system allows the user to also explore the result set using a textual hierarchy which we call Inclusion-DAG. The top picture shows a partially explored Inclusion-DAG of the query "*Bavaria @waterway:waterfall*". The bottom picture shows a close-up of the lowest level where we can still refine the node labeled "Hörnergruppe". The visualization of the Inclusion-DAG has been added as part of [9].

OSCAR tries to fill this gap by enabling a user to search the whole textual data of the OpenStreetMap data set while also adding context-dependent spatial queries. The input to our system is represented by an abstract syntax tree consisting of predicates and boolean operations thereof. As a result we expect all elements in the data set matching the query. The supported predicates encompass substring text content match, polygon intersection tests, and context aware complex spatial relations. We distinguish between two different entities, namely regions and items. The regions are those objects that form large two-dimensional surfaces like a country or a city. Items on the other hand are all other entities like streets, point of interests or smaller areas like a market square. The regions play an important part in our hierarchical result presentation which gives upper bounds for the number of items in each region thus allowing a user the means to explore the result of a query. The semantic data of an object is given by tags as key value pairs. Some of these describe the type of the object, i.e. if it is a recycling bin for clothes, others give a description or a name.

So far we have only discussed aspects of the visualization part of the result. However equally important are the possible types of queries a user can enter. The example mentioned above and displayed in figures 1.1 and 1.2 is a rather simple query: First locate all waterfalls, then remove all that are not in Germany. A very simple implementation using an inverted index may intersect the list of waterfall items with the list of items in Germany without using any kind of geometric information. For many interesting queries it may however be useful to include the geometry as well. In the following we would like to consider a rather advanced usage scenarios that our system should be able to answer. We are interested in a hiking trip and are looking for a nice place to go. We would like to see some tourist attractions while staying at a hotel. We also like to have some restaurants nearby in order to have some culinary variety. A supermarket close by is another welcoming addition in order to resupply for our day trips. Since we don't want to hear any kind of loud noises from driving cars we are interested in a place that is at least 10 kilometers away from any autobahn or trunk road. A further constraint is that we want the place to be somewhere south of Munich. It is a non-trivial task to translate this to a query that we can answer with the search engines displayed in figures 1.1 and 1.2. OSCAR on the other hand is able to calculate the desired place with a single query * in under two seconds. After issuing the query to OSCAR we will find that the result is empty. We either have to travel farther, say south of Bavaria to Tirol, or remove the constraint that we don't want a trunk road nearby since there are just too many in the south. Note that OSCAR is able to handle this type of query on global scale by removing the constraint that we only want to

":south-of #"Munich" (@tourism:attraction %2%%@tourism:hotel %2%%@shop:supermarket %2%%@amenity:restaurant) - (%10%%(@highway:motorway))*"

see results in the south of Munich. Our implementation actually does this since we do not have a query optimizer yet and hence the aforementioned constraint is only applied after retrieving all valid tourist attractions world wide. See figure 1.4 for a visualization of our query.

During the journey to our destination we would like to eat something and get some supplies. This is also possible with OSCAR, albeit with the limitation that we do not have a shortest path calculation yet. Instead the user has to explicitly give a path along which the requested restaurants and supermarkets are to be found. See section 3.4.5 for an extension to OSCAR based on contraction hierarchies to efficiently answer such a query.

Our LGPLv2 licensed source code can be obtained at code.oscar-web.de. Sample data sets are available at data.oscar-web.de.

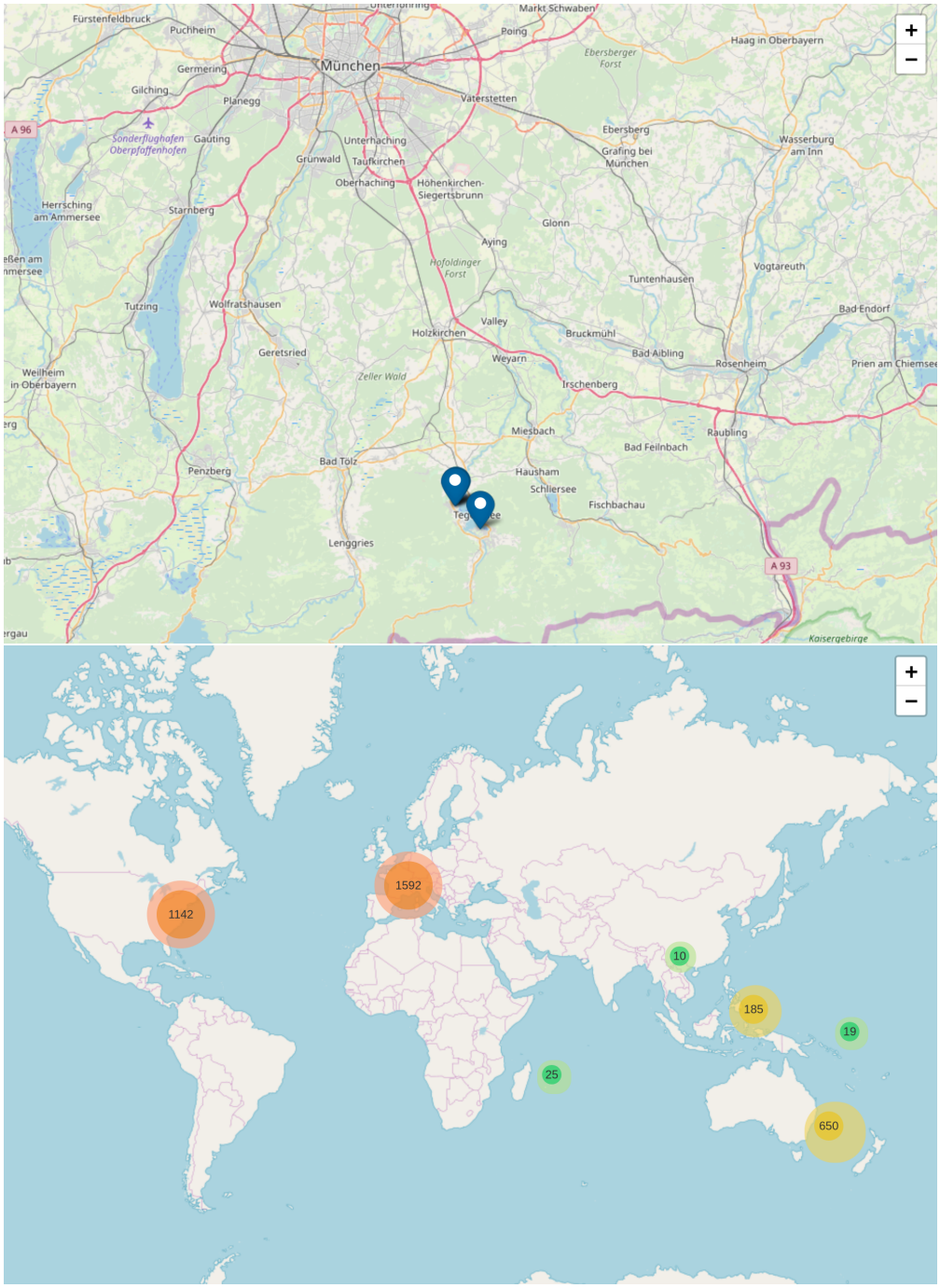


Figure 1.4: Results of our request to display all tourist attractions that are close to a supermarket, restaurant, hotel, are about 10 km away from any motorway and are south of Munich (top) or anywhere on earth (bottom).

1.1 Contribution

In order to be able to answer such complicated queries we first need so solve several problems. We need data structures to efficiently search the large planet data set of the OpenStreetMap project. The results obtained from these structures are used as a starting point for the visualization. The processing step to compute this visualization has to be fast as well in order to interactively explore the result. We therefore chose to develop a data structure that can be used for querying *and* visualization. Probably the most important aspect of OSCAR is that it distinguished between two different entities: regions and items. Regions are two-dimensional surfaces used to represent regional objects like a country, city, suburb or national park. Items on the other hand are all other objects. Regions and items relate to each other in that items are located inside possibly many different regions. If an item is enclosed by a region then we say that this region is a parent of the item. An item may have many parents since multiple regions can cover an item. All items with the same set of parents define a cell. We can now base our text search structures on these cells as follows: We first distinguish between a full-match cell and a partial-match cell. A full-match cell is a cell whose defining items all match the given query whereas a partial-match cell only has some matching items. A single query statement like "*Waterfall*" now returns all the cells containing a matching item which are mostly partial-match cells. If we are interested in waterfalls that are only in Germany then we can query for all cells covered by Germany and intersect this set of cells with the former one. Since the cells of the "*Germany*" query are full-match cells we have that each cell also contains all of the waterfalls. Hence after the intersection of the two sets we know that we now only have the items inside Germany. The crucial point is that there are far less cells than items which speeds-up the computation of the intersection compared to the simple inverted index variant where we would intersect the list of waterfalls with the list of items in Germany. In our comparison with state of the art text indexes, Lucene [1] and MG4J [24], this data structure fared very well with our sample queries gathered from our web-based demonstrator. Additionally we can use this structure to compute our hierarchical visualization: The regions of a data set form a hierarchical structure defined by the inclusion relation. A region r_i includes a region r_j if the set of cells of r_i is a superset of the cells of r_j. This essentially means that r_i completely covers r_j. The regions of a data set thus induce an Inclusion-DAG where we have a node for each region and an edge from nodes r_i to r_j if and only if r_i covers r_j and there is no other region r_k that covers r_j and is covered by r_i. Regions without an incoming edge are connected to a super node. We can now add the cells to this DAG as well using the same inclusion relation: There is an edge from region r_i to $cell_j$ if and only if r_i covers cell c_j and no region r_k exists that covers c_j and r_i. Finally the items are added to the respective

cells. We use this DAG to visualize our result and provide the user a means to interactively explore this DAG. To further improve the exploration we can give for each region an upper bound of the number of matching items that are covered by it. The details are given in section 3.6. Our web based demonstrator, described in section 4.5, uses these counts to draw the cluster markers seen in figures 1.2 and 1.4. A more advanced visualization could use these counts together with the hierarchical information of the Inclusion-DAG to produce a hierarchical heatmap or alternatively a heatmap solely based on the cells. Though for the latter a hierarchy based on a more regular cell structure would be a faster alternative, see section 5.8 for a short description with benchmarks. The text search capabilities together with rudimentary spatial queries have been published in [14]. This work was based on a predecessor of OSCAR, described in [10], where we introduced a storage library which we use to support efficient out-of-memory data structures.

Additional changes to our data structure are necessary to efficiently support complex spatial queries which we introduced in [15]. The regions of our data sets are defined using polygons, usually just one, but multiple polygons defining the interior and exterior are possible as well. Consider for example the mainland of Germany which does not only consist of the large landmass in the middle of Europe but also of a lot of small islands in the North Sea. They are modeled as separate polygons that are also part of the region *Germany*. Sometimes a polygon is necessary to define an exterior, for example to model an exclave. The arrangement of regions is formed by overlaying all segments of the polygons of the regions of the data set while cutting segments at intersection points. A region of space bounded by segments is called a cell. The set of cells induced by the segments of the regions is a refinement of the set of cells mentioned previously. To easily navigate the cells we compute a triangulation for each cell and merge all triangulations of all cells to form one large triangulation. We can now explore the neighborhood of a cell by "walking" in this triangulation. Using this information we can already compute the result of the query for restaurants along a journey if we thread the polygonal line of the journey through the triangulation and gather all cells that we encounter. Additionally we store the direct neighbors of each cell yielding the cell graph. Together with the Inclusion-DAG, which we can interpret as an R-tree, these structures allow us to answer the complicated spatial relation queries. However in order to compute the answer to such a spatial query one first has to define its meaning. Consider for example the two queries "*supermarkets north of Stuttgart*" and "*supermarkets north of the Königstraße in Stuttgart*". Clearly the first query should return supermarkets farther away from Stuttgart than in the second query where we expect results to be close to the Königstraße. We define this context dependence for various complex spatial relations in section 3.3.

Note that in practice we do not calculate the arrangement of segments first but

compute a constraint Delaunay triangulation thereof which gives us the merged triangulation without intermediate steps. One important aspect is that the input of the triangulation are points on the WGS84 [83] geoid. Hence in reality these are actually located in a 3-dimensional space. Our implementation uses a very simple projection to calculate the triangulation in 2 dimensions, see section 4.3.2. This however produces a triangulation that is not Delaunay when mapped back to the geoid. Furthermore intersection points are calculated incorrectly since line segments are Great Circle segments on the geoid. A first steps towards a solution to this problem could be the computation of this triangulation for the points mapped on the unit sphere. However in section 6.2.1 we prove that it is not possible to represent the projected input points using floating-point numbers alone. Instead one has to use rational numbers in order to represent points that are directly on the unit sphere. One may wonder if such great measures have to be taken in order to compute a triangulation of input points that are the result of error-prone measurements. The unfortunate answer is that algorithms may return false results or even crash if the used predicates are evaluated incorrectly due to the test points not being exactly on the sphere. Since the input points result in irrational numbers but we are constraint to rational numbers we can only compute an approximation of the real triangulation based on rational approximations of the real three-dimensional coordinates. A method to compute such an approximation was described in [101]. However an implementation of the algorithm is rather difficult and likely far from efficient. We introduced the idea for a simpler algorithm in [13] with a full description and analysis in higher dimensions in [17, 16] which we reproduce in part in chapter 6. We show how the use of rotation symmetry and approximations with fixed-point numbers suffice to improve on the main theorem of [101]. With our method it is possible to derive rational points *exactly on* $\mathbb{S}^{d-1}$ with denominators of at most $10(d-1)/\varepsilon^2$ for any $\varepsilon \in \left(0, \frac{1}{8}\right]$. Moreover, it is possible to compute even smaller denominators based on algorithms for simultaneous Diophantine approximations, though a potentially weaker form of approximation would suffice. Our approach allows for inexact but ε-stable geometric constructions – e.g. the aforementioned intersections of Great Circle segments. The practicability of our algorithm is demonstrated in an extensive experimental evaluation on a planet scale data set.

Prior Publication We introduced our serialization library together with an early version of OSCAR in [10]. OSCAR itself has seen its light in [14] where we describe the general text search idea with support for simple rectangular spatial constraints. We extended OSCAR with complex relational queries in [15] based on a triangulation and an improved web-based demonstrator. The work on rational points on the unit sphere began in [13] where we described our construction algorithm. A first analysis together with an extension to general dimensions was published in [17] with an improved analysis in [16]. We explicitly state for respective sections, subsections, paragraphs or figures if they have been published before.

1.2 Related Work

Textual-Spatial search is a well researched field with many different applications. Since a data structure tailored to a specific problem is almost always better than a generic variant one will find many possible solutions for the many different applications. In order to be able to classify OSCAR we first like to give a simplified overview of this vast research field categorized by query type and result presentation. In its simplest form we are only interested in text search or spatial search.

1.2.1 Text Search

In text search one is usually interested in matching an input string, the pattern, against another string, a set of strings or sets of strings. Suffix search data structures for single strings are especially useful in genome analysis. One of its simplest form is the suffix array [78] allowing substring search. Its space usage grows linear in the size of the original string with each entry taking $\log_2(n)$ bits. Since genomes have a rather small alphabet this may result in quite a large data structure compared to the input data where a character occupies only 2 bits. To alleviate this various compressed variants where proposed [52, 99, 53] with the FM-Index [40, 41] being one of the first to open the field of compressed self-indexes. These are text indexes that need space close to the entropy of the input text while also storing the text itself. Further research focused on finding compressed variants of the suffix trie [100] which allows a user to efficiently navigate the trie of suffixes compared to the simple suffix array which is just a sorted array of the suffixes. Matching multiple strings is also possible by concatenation of the input strings with a special delimiter for each string [40].

Document Retrieval However taking this even further to support finding objects by their set of strings while supporting boolean queries is out of scope. We will refer to this special type of query as document retrieval with the objects being documents of a larger document collection. A document usually consists of multiple fields among them a title and the body. The text of the fields is then split into chunks, usually at word boundaries, also called tokens. The search structure is built based on these tokens. Its simplest form is the inverted index where we store for each token the set of documents containing that token. These sets are also called posting lists and usually reference the documents by a unique number. More advanced solutions combine entries to support faster boolean queries [21], or support more advanced queries like phrase queries or proximity queries where we want multiple query strings to be close together in the documents text [24]. Ranking the result of a query is an important task which may also rely on extra information stored in our indexing structure. Ranking functions may take into ac-

count the importance of words, the quality of the match or the aforementioned complex queries like the order of the words. The way the posting list is stored has a tremendous impact on the performance of these data structures which is why there is extensive research on how to store, retrieve and operator on compressed variants of the posting lists. Depending on the nature of the list various optimizations are possible some of which we will introduce in section 2.4. There exist numerous full-featured frameworks among them ATIRE [106], Galago [71], Lucene [1] and MG4J [24] that support the whole process of storing and retrieving documents of a collection. Most systems employ compression techniques like variable byte encoding after delta-compression - MG4J being one of the few using a quasi-succinct index to store posting lists [109]. Seven open-source text retrieval frameworks were compared in [75] using various text collections where MG4J used the least amount of storage while having competitive query times.

Approximate String Matching So far we have only introduced data structures allowing prefix or substring matching. However it may also be of interest to approximately match the pattern with the text, that is we want to allow errors in spelling. Most practically relevant algorithms employ suffix tries or q-gram based data structures in one way or another. In its simplest form one uses backtracking to explore the suffix trie. [84] classify various practically working approaches based on the search approach, if errors are in the text or the pattern and the data structure it is based upon.

1.2.2 Spatial Search

Spatial search with regards to OSCAR is interested in finding spatial objects intersecting a given region of interest. Additionally one may also be interested in the k nearest neighbors of a given object or the top-k objects subjected to some user-define measure and intersecting the query region. In its simplest form we have a set of points and are interested in all points with a non-empty intersection with an axis aligned bounding box. Solutions to this problem include the range tree, kd-tree or other binary space partitioning schemes. For line segments one can use the segment tree. However some of these data structures have a storage need of $\mathcal{O}(n \log n)$ for two-dimensional data sets. Additionally integrating objects of different types is a non-trivial extension. Hence in practice one usually finds space partitioning schemes that are either object-oriented like the R-tree [55] or space oriented like a quadtree [42]. The R-tree comes in many flavors that differ in the used construction algorithm [22, 63], combinations with other search structures or other changes in one of the many parameters. In some variants it is even allowed to put objects into multiple leaf nodes [102]. Yet they all share the same underlying structure: the objects are recursively grouped together forming a tree

structure. The query performance heavily depends on the data set and the query [59].

Quadtrees are an incarnation of hierarchical space portioning schemes. Among them is also Kirckpatrick's hierarchy [64], which is based on a triangulation or the HTM index which we cover in short in section 2.3.1. With regards to geospatial data the OGC Discrete Global Grid System core standard [94] is of great interest since it demands among other things that cells in the same level of the hierarchy should be of equal area and form.

1.2.3 Textual Spatial Search

In Textual Spatial Search we are interested in the combination of the aforementioned search capabilities. To be precise we would like to retrieve all items of a data base subject to a textual match *and* a spatial match. The commonly used approaches to solve this problem either use separate indexes, a spatial index as primary structure, a text index as primary structure or interleave text and spatial search structures.

Separate Indexes The separate index approach, for example as in [107], uses any of the aforementioned textual and spatial indexes separately and intersects the results returned by the two search data structures.

Spatial Primary Indexes The spatial structure first approaches combine a spatial search data structure as primary data structure with a text search structure as secondary search structure. A simple variant would be a grid where each cell has an inverted index for the items intersecting the grid cell (e.g. [107]). More elaborate variants combine a quadtree or an R-tree with an inverted index [114].

Text Primary Indexes The text structure first approaches on the other hand do the reverse and use a text search structure as primary data structure. As before a simple variant would be an inverted index where each entry stores a grid containing the items of the entry's posting list. Improved variants may use a quadtree (e.g. [113]) or an R-tree (e.g. [114]) as secondary search structure.

Interleaved Indexes A natural third approach would be a data structure interleaving a text search and spatial search structures. Variants based on R-trees are described in [58, 74, 39, 33] which are based on the idea of pruning subtrees not containing items matching the query string while obeying the spatial constraint using the information provided by the R-tree structure. To this end they store information about the textual content in the inner nodes of the R-tree. Additionally

some variants store information needed for ranking the result thus pruning even more subtrees if only the top-k items are of interest.

[73] propose the so-called MHR-tree that allows to retrieve all items within a given query rectangle and (approximately) matching a set of given strings. Their data structure is a mixture of an R-tree and a q-gram based text index. Each node of the R-tree stores a min-wise signature of the q-grams of the strings of the items that are part of the subtree of the node. Given a query string and such a min-wise signature it is possible to estimate whether there are hits in a subtree of the respective node. This information is then used to prune nodes that do not contain items relevant to the query. However it is unclear how well their scheme works on the data set handled by OSCAR given that their benchmarks only consider single states in the US. Additionally items likely only store a small number of strings. Thus one would need to increase the size of the signatures in order to keep the pruning capabilities of the system at the same quality when adding more strings to the items.

As an alternative to R-trees one may also use a quadtree as spatial search structure as proposed in [110].

A more in-depth overview of recent advances in textual spatial indexing can be found in [95].

Unfortunately almost none of the aforementioned works provide a publicly available implementation which is why we cannot compare them with OSCAR on the same large data set without implementing them ourselves. Additionally the largest data sets used in benchmarks are up to two magnitudes smaller than what OSCAR needs to handle.

1.2.4 Directional relations [1]

Directional relations are frequently used to select data in spatial databases and are fundamental to spatial data queries, analysis and reasoning [6, 50]. Directional relations are not only used widely in geographic information systems, but also in areas like artificial intelligence [66], computer vision [80], and multimedia [112]. Consequently there has been a significant amount of effort to determine directional relations automatically.

A commonly used indicator for the directional relation between two regions is the direction between their centroids [88], possibly snapped to one of the 8 compass directions. As an advantage, this indicator is symmetric and once the centroids are (pre)computed, efficient to evaluate for a pair of regions. Yet, sometimes asymmetric answers make more sense. The popular direction-relation matrix model [49] allows for such asymmetric answers. It subdivides the space around the bounding box of the reference region into nine direction tiles and classifies other regions according to the cell of the subdivision they lie in. [27] introduce a splitting line

model to decide on the relation of two geometric entities. The goal of all these approaches is to compute a directional indicator between two given regions, whereas for our concrete application we want to determine regions fulfilling a certain directional relation. In particular we also need to derive the *extent* of a region for the matches, not only a direction.

1.2.5 Result Presentation

So far we have only introduced data structures and algorithms that return elements matching a given query. However equally important is an efficient and effective visualization of the result. Efficient in the sense that it is cheap to compute and effective in the sense that it helps a user in understanding the result. The probably most widely used visualization scheme is based on a result list in conjunction with icons for each item in the list as displayed in figures 1.1 and 1.2. If the result set is too large then only a number of items is displayed and further items can be loaded on user request which happens either implicitly (continuous scroll) or explicitly (pagination, button to load more). [67] evaluate the impact of the visualization on user satisfaction in the context of multi-criteria local search. Specifically they compare a standard icon-based approach with a heatmap-based approach and find that the two perform equally well. However ten out of the fifteen participants stated that a mixed visualization that uses heatmaps to get an overview and icons for a closer more detailed look were preferable. Figure 1.5 shows the system described in [8] which is a powerful web-based data analysis tools supporting various visualizations among them heatmaps and choropleth maps with many user configurable parameters. The system allows a user to upload data which is then analyzed on a server and send back for visualization. The visualization itself is mostly written in JavaScript with the back end written in R. Unfortunately the authors do not publish any performance analysis which is why we cannot assess whether the system could handle the large query results that OSCAR may produce. The example files on their website are rather small hence we assume that their system likely does not cope very well with very large data sets.

[7], shown in figure 1.2 describe an exploratory search engine for thematic mapping which has many similarities to OSCAR. They also distinguish between regional entities and smaller items. Their data structure is mainly based on a hierarchical discrete global grid. Each regional entity and item is assigned to the grid cells it intersects. A document in their system may then reference regional entities and items with a weight assigned to the reference. The set of documents referencing a grid cell then forms a grid document which is indexed using standard inverted indexes. On query time the system retrieves the grid cells matching the query string as well as the spatial constraint. The level of the retrieved grid cells depends on the zoom level of the user. The result is displayed using a

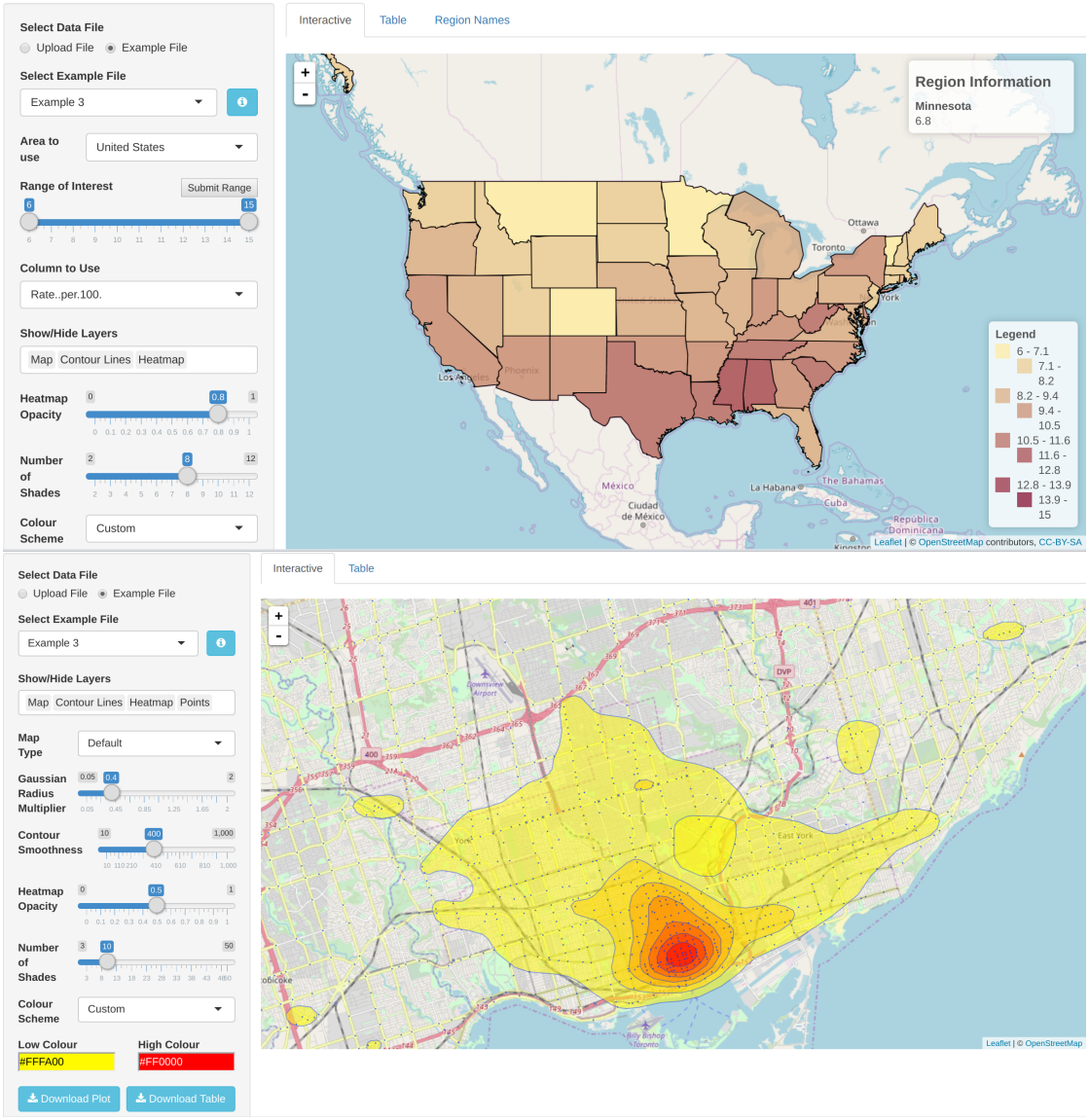


Figure 1.5: Heatmapper's example visualizations. The top shows a choropleth map whereas the bottom shows a heatmap with contour lines. On the left sides are the controls with which the user can change the parameters of the visualization.

heatmap with colors based on the importance of the cells which is calculated using the weights assigned to the individual documents. Together with the zoom level depended grid level selection this has the nice property that on higher zoom levels entities with a larger extend are more important than very small ones and thus giving a good overview of a result. A user can then click on a cell and view the top-k results of that cell. Compared to OSCAR the system lacks the explicit semantic hierarchy with information about the number of matched results and the complex relational queries. Additionally the performance of their system is unclear since they do not report any data regarding the size of the index or the speed of the retrieval process.

1.2.6 Classifying OSCAR

The previous sections introduced various classifying schemes among different aspects of an information retrieval system. Regarding the search structures OSCAR uses a text first approach with an suffix array as text index and a triangulation as secondary spatial search structure. However we do not store the secondary search structure explicitly but rather partition the posting lists of the text index according to the spatial index. Approximate spatial constraints are then computed separately using another data structure with which we select the appropriate grid cells. Thus most computations are based on simple integer set operations. Additionally complex operations like the dilation operator are easy to implemented. Our result presentation is a mixture between choropleth maps and item based visualization. We augment the choropleth maps with cluster icons for higher zoom levels in order to get an overview of the result data and switch to an icon based variant on lower zoom levels letting a user inspect the result in more detail.

1.2.7 Rational Points on the Unit Sphere [2]

Studies on spherical Delaunay triangulations (SDT), using great-circle segments on the sphere $\mathbb{S}^2$, provide common ways to avoid and deal with the point-on-sphere problem in computational geometry.

The *fragile* approaches [90, 61, 96] ignore that the input may not be on $\mathbb{S}^2$ and succeed if the results of all predicate evaluations happen to be correct. Input point arrangements with close proximity or unfortunate locations bring these algorithms to crash, loop or produce erroneous output. The *quasi-robust* approaches [26, 20] weaken the objective and calculate a Delaunay tessellation in d-Simplexes. Lifting to a $d+1$ convex hull problem is achieved by augmenting a rational coordinate from a quadratic form – The augmented point exactly meets the (elliptic) paraboloid equation. However, the output only identifies a SDT if all input points are already on the sphere, otherwise the objectives are distinct. Equally

unclear is how to address spherical predicates and spherical constructions. The *robust* approaches [98] use the circle preserving stereographic projection from $\mathbb{S}^2$ to the plane. The perturbation to input, for which the output is correct, can be very large as the projection does not preserve distances. Furthermore, achieving additional predicates and constructions remains unclear. The *stable* approaches provide geometric predicates and constructions for points on $\mathbb{S}^2$ by explicitly storing an algebraic number, originating from scaling an ordinary rational approximation to unit length[34]. Algebraic number arithmetics can be avoided for $\mathbb{S}^2$, but exact evaluation relies on specifically tailored predicates [30], leaving the implementation of new constructions and predicates open.

Kleinbock and Merrill provide methods to quantify the density of rational points on $\mathbb{S}^d$ [65], that extend to other manifolds as well. Recently, Schmutz[101] provided an divide-&-conquer approach on the sphere equation, using Diophantine approximation by continued fractions, to derive points in $\mathbb{Q}^d \cap \mathbb{S}^{d-1}$ for a point on the unit sphere $\mathbb{S}^{d-1}$. The main theorem bounds the denominators in ε-approximations, under the $\| \ \|_\infty$ norm, with $(\sqrt{32}\lceil \log_2 d \rceil / \varepsilon)^{2\lceil \log_2 d \rceil}$. Based on this, rational approximations in the orthogonal group $O(n, \mathbb{R})$ and in the unitary matrix group $U(n, \mathbb{C})$ are found. This is of particular interest for sweep-line algorithms: [29] studies finding a rotation matrix with small rationals for a given rational rotation angle of an 2D arrangement.

1.3 Outline

We first introduce some basic data structures and algorithms on which OSCAR is based upon in chapter 2. A formal definition of OSCAR's data structure and its supported query types follows in chapter 3. Chapter 4 deals with the concrete implementation of these data structures. An extensive experimental evaluation in chapter 5 shows OSCAR's efficiency. We also compare our approach with state of the art text indexes and show how the chosen cell arrangement impacts the storage consumption and query times. In chapter 6 we give our method to compute rational approximations of points on the unit sphere. We conclude with a short summary of our work and an outlook on future improvements in chapter 7.

2 Preliminaries

In this chapter we would like to introduce the reader to some basic data structures needed to understand the inner workings of OSCAR. We start with an introduction to OpenStreetMap and its data model. We then proceed to discuss some possible data structures for text and spatial search. We also give short description of commonly used compression schemes for lists of integers. Finally we give the basics necessary to understand our approximation method to compute rational points on the unit sphere.

2.1 OpenStreetMap

OpenStreetMap [2] is a collaborative crowd-sourced mapping effort to produce a free map of the world. It provides very detailed data in most developed countries with thousands of daily users [4]. OpenStreetMap's data model distinguishes three different data types: nodes, ways and relations. A node references a point on the WGS84 reference ellipsoid. Ways are defined by an ordered list of nodes whereas relations consist of nodes, ways and relations. A node is usually used to represent a point of interest or objects of small size. Ways are used to model polygonal chains like streets or simple polygons like buildings. Relations are used in many different scenarios such as modeling the border of a nation state, combining several ways to represent hiking routes or to define the street a building is associated with. Tags in the form of key-value pairs are used to define the semantic of an element (see figure 2.2 for some examples). Tags may be of direct use to a user, like a name of an entity, or only indirectly like the *building* key (compare with figure 2.1). In the following we will refer to those tags whose value are directly interpretable as the *important* tags. Among them are the values of the *name* key or the values of the *addr* key.

We furthermore treat specific regional entities like nation states and national parks in a special way. In the following we will call these entities regions whereas all other entities are called items. Examples of these are depicted in figure 2.3.

Data The data of the OpenStreetMap project is stored in a SQL database. It is exported either as hourly difference files or as weekly complete database dumps. This data set is usually referred to as the "planet" data set. There are also subsets of this data available for example from [5]. There are generally two different data formats in wide spread use: An XML format that is easy to read and parse and

a binary format based on the Google Protocol Buffers format specification[3]. As of 2017 the XML format takes about 900 GB of storage space whereas the binary format only needs 40 GB.

There is a rich ecosystem of tools centered around the processing of OpenStreetMap data files. We use the library *osmpbf* [54] to parse the binary files.

key	value
addr:city	Stuttgart
addr:country	DE
addr:housenumber	5
addr:postcode	70173
addr:street	Schulstraße
amenity	fast_food
brand	McDonald's
cuisine	burger
email	mgr60327@store.de.mcd.com
internet_access	wlan
internet_access:fee	no
internet_access:operator	Telekom
name	McDonald's
phone	+49 711 292370
smoking	no
website	http://www.mcdonalds.de
wheelchair	no

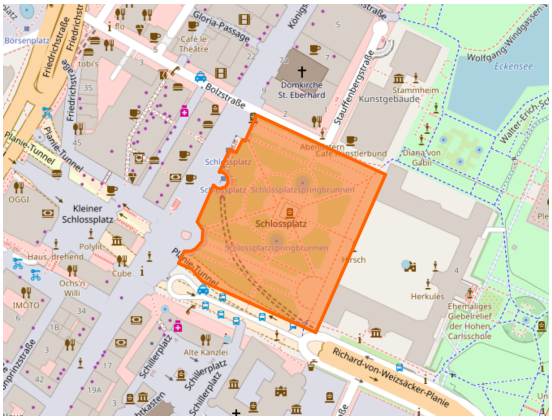
Figure 2.1: Complex example of a McDonald's in Stuttgart as described in OpenStreetMap.

	type	attribute
doctor's office	node	amenity=doctors
pharmacy	node	amenity=pharmacy
autobahn	way	highway=motorway
building	way	area=yes building=yes
hiking route	relation	type=route, route=hiking
country border	relation	type=boundary, boundary=administrative

Figure 2.2: Example tags in OpenStreetMap

```
<node id="3671613521"
      lat="48.7748479"
      lon="9.1779891">
  <tag k="description:en"
       v="Unusual contiuous loop elevator: \
          hop on hop off."/>
  <tag k="indoor"
       v="yes"/>
  <tag k="name"
       v="Rathaus Paternoster"/>
  <tag k="tourism"
       v="attraction"/>
  <tag k="wheelchair"
       v="no"/>
</node>
```


```
<way id="3933618">
  <nd ref="2141764850"/>
  ...
  <nd ref="2141764850"/>
  <tag k="alt_name" v="Schloßplatz"/>
  <tag k="internet_access" v="wlan"/>
  <tag k="name" v="Schlossplatz"/>
  <tag k="place" v="square"/>
  <tag k="toilets:wheelchair" v="no"/>
  <tag k="tourism" v="attraction"/>
  <tag k="url"
      v="http://stuttgart.de/
         item/show/305802/1/dept/108937?"/>
  <tag k="wheelchair" v="limited"/>
  <tag k="wikidata" v="Q242067"/>
  <tag k="wikipedia"
       v="de:Schlossplatz (Stuttgart)"/>
</way>
```




```
<relation id="1107850"">
  <member type="way" ref="330743395"
          role="outer"/>
  ...
  <member type="way" ref="70547264"
          role="outer"/>
  <tag k="admin_level" v="9"/>
  <tag k="boundary" v="administrative"/>
  <tag k="name" v="Stuttgart-Mitte"/>
  <tag k="name:prefix" v="Stadtbezirk"/>
  <tag k="type" v="boundary"/>
  <tag k="wikidata" v="Q727750"/>
</relation>
```

Figure 2.3: Items and regions, their associated data in XML format and the way they are rendered by the OpenStreetMap website. At the top is the Paternoster in the town hall in Stuttgart modeled as a node. The middle picture shows the Schlossplatz modeled as a closed way which we may store as either an item or a region and finally at the bottom we see the city district Stuttgart-Mitte of Stuttgart. Both the Paternoster and the Schlossplatz are covered by this region. If the Schlossplatz is defined to be a region as well then it would be a child of Stuttgart-Mitte.

2.2 Text Search

In text search one has as input the text to be searched and a pattern that we are looking for. Usually the text is fixed but the pattern may change. Hence we are interested in data structures that support fast pattern matching for a given text. Additionally we may be interested to search multiple texts for the same pattern. This problem is also called document retrieval where we have a collection of documents and a pattern and we would like to get a list of documents matching that pattern. Usually a document is split into fields like the title and its content. A field is again split into tokens which is the data that we want our pattern to match against. One simple data structure to find all matching documents is the inverted index. An inverted index is a sorted array of the tokens where each entry has a list of documents containing that token. Searching for a prefix is easily accomplished with a simple binary search. If we are interested in suffixes or substring matches then we can build an inverted index for all suffixes of the input tokens. Again a simple binary search suffices to answer a query. If we did a prefix search on this structure then we would answer a substring query. As of now a prefix search would return a range of matching entries. It may however be beneficial to only return a single entry. We can achieve this by constructing the array based on all possible substrings. This array will be really large but allows us to answer substring queries in an instant. We can compress this array by merging neighboring entries if they point to the same set of source documents and one is a prefix of the other. We simply keep the longer entry and discard the shorter one. What we then get is the suffix tree, which is a trie of the suffixes, encoded in the array with explicit internal nodes. The entries of this structure likely have many equal document sets which we can store in an extra data structure and simply point to them. See figure 2.4 for an example.

Documents

Id	Content
0	berghof
1	feldbergen
2	mittelberg

Posting lists

Id	Documents
0	0
1	0, 1
2	0, 1, 2
3	1
4	1, 2
5	2

Invertex Index

String	Length	Documents	Posting list
berg	4	0, 1, 2	2
bergen	6	1	3
berghof	7	0	0
dbergen	7	1	3
e	1	0, 1, 2	2
el	2	1, 2	4
elberg	6	2	5
eldbergen	9	1	3
en	2	1	3
erg	3	0, 1, 2	2
ergen	5	1	3
erghof	6	0	0
f	1	0, 1	1
feldbergen	10	1	3
g	1	0, 1, 2	2
gen	3	1	3
ghof	4	0	0
hof	3	0	0
ittelberg	9	2	5
l	1	1, 2	4
lberg	5	2	5
ldbergen	8	1	3
mittelberg	10	2	5
n	1	1	3
of	2	0	0
rg	2	0, 1, 2	2
rgen	4	1	3
rghof	5	0	0
telberg	7	2	5
ttelberg	8	2	5

Figure 2.4: An inverted index supporting substring search. Posting lists are stored with ascending document ids.

2.3 Geometry Search

2.3.1 Space partitioning

A space partitioning is a division of space into disjoint subsets. The simplest form being a grid where one divides the space into equal sized rectangular cells. An improvement to this are quad-trees where we recursively divide a cell of the grid into smaller cells using a smaller grid. This is usually done by splitting the cells into 4 smaller cells – hence the name quad-tree. Another approach are Binary space partitioning schemes where the space is divided by a single hyperplane. The hyperplanes may be positioned arbitrarily (BSP-Tree) or along the coordinate axis (Kd-Tree).

Discrete Global Grid Systems provide a hierarchical subdivision of the unit sphere into cells. One possible implementation is the aforementioned quad-tree. However this produces cells that are not equal in size or form in each level. A variant based on triangles with the property that cells within the same level are approximately equal size and form was introduced in [104]. Many other variants are possible as well, for example one based on hexagons implemented by [60]. The Open Geospatial Consortium defines an abstract standard [93] for discrete global grid systems. The standard defines operations and properties that grid systems should provide. We are especially interested in point location, neighborhood exploration and cell addressing.

Triangulations

Triangulations partition the space using only simplexes – in our case only triangles. Assuming non-degeneracy the Delaunay triangulation of a set of points is uniquely defined. For a given triangle its circumcircle is the circle passing through its 3 vertexes. In a Delaunay triangulation we have that the circumcircle of each triangle only contains the 3 vertexes of its defining triangle and no other vertex of the triangulation. Furthermore this maximizes the minimum angle of all triangles and thus produces ”nice” looking triangulations. We can extend this triangulation to also support constraints. These are edges that have to be part of the triangulation. This type of triangulation is also called constrained Delaunay triangulation and is a triangulation that is as close to a Delaunay triangulation as possible while containing the aforementioned set of constrained edges. See figure 2.6 for an example. We can transform a given triangulation to a Delaunay triangulation as follows:

1. Find a quadrilateral based on two adjacent triangles not adhering to the circumcircle property

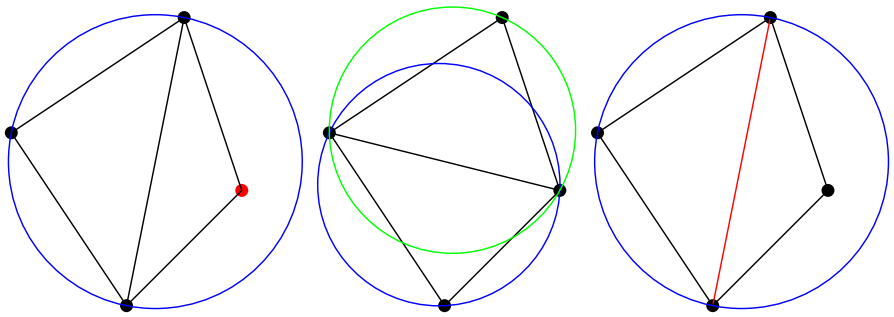


Figure 2.5: Left: A quadrilateral with a point violating the circumcircle property. Middle: Flipping the edge shared by the two triangles produces a valid quadrilateral. Right: Constrained edges are not allowed to be flipped, thus producing triangles with points violating the circumcircle property.

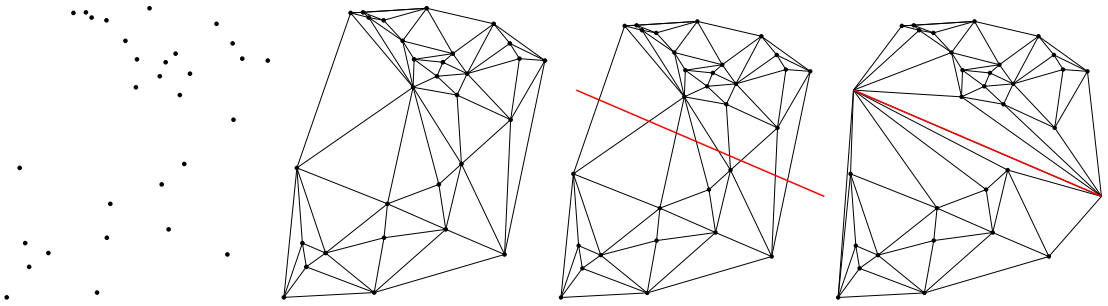


Figure 2.6: A Delaunay triangulation of a set of points. We then add a constraint edge (red) and get a constrained Delaunay triangulation.

2. Flip the edge which the two triangles share if it is not a constrained edge

We are finished if we do not find any violating quadrilateral anymore with an edge that can be flipped See figure 2.5 for an example.

2.3.2 Bounding volumes

A bounding volume is a simple geometric object used to approximate the region of space of a given set of shapes. For example axis-aligned bounding boxes represent the set by a single rectangle. Depending on the form of the shapes a sphere like the minimum enclosing ball may be a better approximation. Another option is to use the convex hull of the shapes which is the smallest region of space such that every line segment with endpoints on any of the shapes is inside that region. See figure 2.7 for examples. We may use multiple bounding volumes per object to

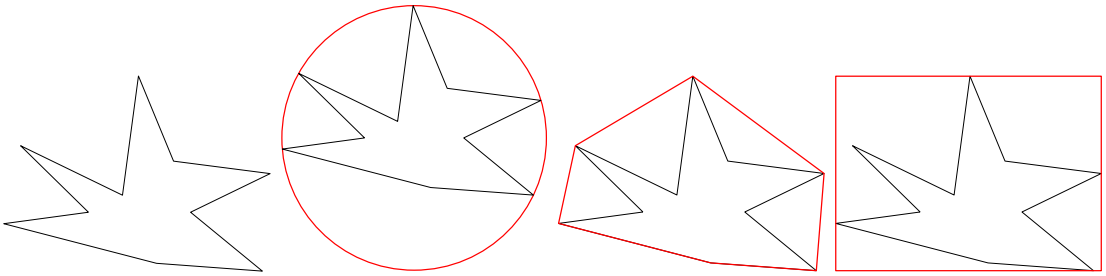


Figure 2.7: A shape (black) with bounding volumes (red) from left to right: sphere, convex hull and rectangular bounding box

further speed up geometric predicates like the intersection test. For rather large sets of objects it is advisable to split the set into multiple subsets where each is approximated with a bounding volume. We get a bounding volume hierarchy if we do this recursively. Depending on the choice of bounding volumes and the way we group objects we may for example get an R-Tree. The R-Tree uses a rectangle as bounding volume and objects are grouped according to multiple strategies. One simple heuristic is to group nearby objects together with a predefined minimum and maximum number per group. See figure 2.8 for an example.

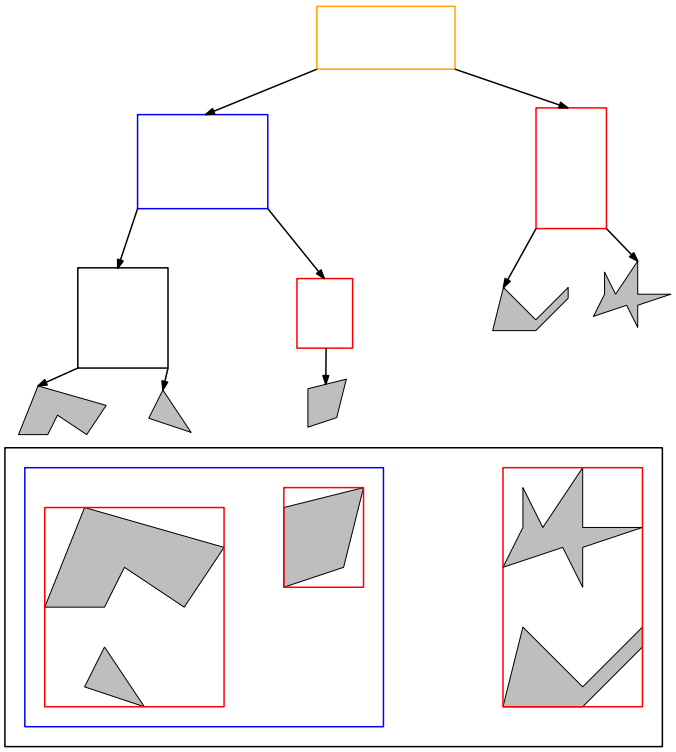


Figure 2.8: A bounding volume hierarchy with 5 shapes and rectangles as bounding volumes essentially forming an R-tree.

2.4 Compression techniques

There is a vast body of work on compression techniques which we cannot cover in any meaningful detail which is why we refer the reader to the respective literature. However we would like to present some techniques to compress sequences of integers since these will be used in many places in our system.

Delta and run-length encoding Delta encoding is probably the first improvement to encode monotone sequences of integers. Instead of storing each integer we can store the difference to the previous entry. If there are a lot of equal numbers then a run-length encoding may reduce the storage need by encoding these runs of multiple numbers in only two numbers. This is especially useful in conjunction with the aforementioned delta encoding. See table 2.1 for an example.

So far we have not talked about how to store the numbers mentioned before. If we know an upper bound U to the sequence then we can encode each entry in $\log_2 U$ bits.

Universal codes If no upper bound is known beforehand then we have to fall back to universal codes like Elias codes [37]. The simplest version being the Elias gamma code where one first encodes the magnitude of the number in unary and then appends the number in binary. If we apply this transformation to the unary part then we get the Elias delta code. The Elias omega code recursively applies this transformation to further reduce the amount of storage needed to encode really large numbers.

However in the context of this work we may assume that no number is larger than 2^64 – in our implementation we assume an even lower upper bound of 2^32. Hence let us consider how to store these bounded integer sequences.

VByte encoding One simple solution is variable byte-length encoding where we split the number into chunks of 7 bits. Each chunk is stored in a byte where the 8th bit is used to indicate if another byte follows that is part of the number.

Raw	0	23	23	42	42	42	43	44	45	46	47	48	49	50
Delta	0	23	0	19	0	0	1	1	1	1	1	1	1	1
Run-length	0	$2 \cdot 23$		$3 \cdot 42$			43	44	45	46	47	48	49	50
Run-length Delta	0	23	0	19	$2 \cdot 0$		$8 \cdot 1$							

Table 2.1: Example of delta and run-length encoding applied to monotone sequence of integers

Varint encoding In the Varint-GB scheme [35] we encode blocks of 4 Integers with a control byte in front which indicates for each of the four numbers how many bytes that number needs. An improved version called Varint-G8IU [103] can be decoded with SIMD instructions. In this scheme the size of a block is fixed to 9 bytes and we encode 1 to 8 numbers per block. Again the first byte is the control byte which defines the structure of the following 8 bytes. See [69] for a nice explanation of these schemes together with an improved encoding scheme called Stream VByte based on the Varint-GB scheme: Control bytes are stored at a different memory location and thus have predictable addresses which is more SIMD friendly.

Frame of Reference encoding Taking this a step further by increasing the block size will lead us to Frame of Reference (FoR) coding. We can encode the elements of a block in relation to the minimum element m and maximum element M within the block. To this end we store m explicitly and all other elements as difference to the minimum element using only $\log_2 M - m + 1$ bits per entry. A block encoded like this is called a frame of reference [47]. Consider the block of length 1024 consisting of 1020 ones and 4 twos. Clearly using 2 bits per entry is worse than using 1 bit per entry while storing the twos explicitly somewhere else. We essentially patch the frame of reference to not include the 4 bad entries – hence the name Patched Frame of Reference (PFoR) [115]. There is a multitude of options to store these outliers with different trade offs regarding compression speed, decompression speed and compression ratio. Various (P)FoR compression schemes were compared in [31] where simple FoR was among the best performing compression schemes regarding the time space trade-off.

2.5 Spherical Geometry [3]

2.5.1 From Spherical Coordinates to Cartesian Coordinates

When working with spherical coordinates it is often necessary to convert them into Cartesian coordinates. We can assume that the input is given in degrees and hence as multiples of π with two coordinates θ and ϕ where θ is the polar angle and ϕ the azimuthal angle. The Cartesian coordinates are then given by:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} sin(\theta) \cdot cos(\phi) \\ sin(\theta) \cdot sin(\phi) \\ cos(\theta) \end{pmatrix} \tag{2.1}$$

Algebraic representations The 2nd *Chebyshev polynomials* U_n of degree n are in $\mathbb{Z}[X]$, given their recursive definition:

$$\begin{aligned} U_0(x) &= 1 \qquad U_1(x) = 2x \\ U_{n+1}(x) &= 2xU_n(x) - U_{n-1}(x) \quad . \end{aligned}$$

It is well known [97], that the n roots of U_n are exactly the values

$$\{\ \cos\left(\pi k/\left(n+1\right)\right)\ :\ k = 1, \ldots, n\ \}$$

. Hence the polynomials U_n give rise to algebraic representations for cosine values of rational multiplies of π.

2.5.2 Stereographic Projection

Let $p = (0, \ldots, 0, 1) \in \mathbb{R}^d$ be the fixed point for the projection τ, mapping all points of a ray from p to the intersection with the hyperplane $x_d = 0$.

$$\begin{aligned} \tau : \mathbb{R}^d \setminus (\mathbb{R}^{d-1} \times \{1\}) &\to \mathbb{R}^{d-1} \\ x &\mapsto \left(\frac{x_1}{1-x_d}\ ,\ \ldots\ ,\ \frac{x_{d-1}}{1-x_d}\right) \end{aligned}$$

The surjective mapping τ is injective as well, when restricted to the domain $\mathbb{S}^{d-1} \setminus \{p\}$. We further define the mapping σ, which is

$$\begin{aligned} \sigma : \mathbb{R}^{d-1} &\to \mathbb{R}^d \setminus \{p\} \\ x &\mapsto \left(\ \frac{2x_1}{1+S^2}\ ,\ \ldots\ ,\ \frac{2x_{d-1}}{1+S^2}\ ,\ \frac{-1+S^2}{1+S^2}\ \right) \end{aligned}$$

where $S^2 = \sum_{j=1}^{d-1} x_j^2$. We have $\operatorname{img}\sigma \subseteq \mathbb{S}^{d-1}$, since

$$\|\sigma(x)\|_2^2 = \frac{(-1+S^2)^2 + \sum_{i=1}^{d-1}(2x_i)^2}{(1+S^2)^2} = 1 \quad .$$

Furthermore, $x = \tau \circ \sigma(x)$ for all $x \in \mathbb{R}^{d-1}$, since

$$\left(\tau \circ \sigma\right)_i(x) = \frac{\frac{2x_i}{1+S^2}}{1 - \frac{-1+S^2}{1+S^2}} = \frac{2x_i}{1+S^2+1-S^2} = x_i$$

holds for all $1 \leq i < d$. Hence, σ and τ are inverse mappings. Note that images of rational points remain rational in both mappings, establishing a bijection between rational points in $\mathbb{R}^{d-1}$ and $\mathbb{S}^{d-1}$.

2.5.3 Diophantine Approximation

The field of Diophantine approximations studies how well a given real number can be represented by a rational number. In this context a good approximation is one with a small denominator. We want to give two important results which we will need later.

Theorem 1 (Liouville's Lower Bound). *For any algebraic $\alpha \in \mathbb{R}$ of degree $n \geq 2$, there is a positive constant $c(\alpha) > 0$ such that*

$$\left|\alpha - \frac{p}{q}\right| \geq \frac{c(\alpha)}{q^n}$$

for any $p \in \mathbb{Z}$ and $q \in \mathbb{N}$.

Apart from this lower bound on rational approximations, there is another important folklore result on the existence of simultaneous Diophantine approximations. Such approximations have surprisingly small errors, despite their rather small common denominator.

Theorem 2 (Dirichlet's Upper Bound). *Let $N \in \mathbb{N}$ and $\alpha \in \mathbb{R}^d$ with $0 \leq \alpha_i \leq 1$. There are integers $p \in \mathbb{Z}^d$, $q \in \mathbb{Z}$ with $1 \leq q \leq N$ and*

$$\left|\alpha_i - \tfrac{p_i}{q}\right| \leq \frac{1}{q\sqrt[d]{N}} \ .$$

For $d = 1$, the continued fraction (equivalently the Euclidean) algorithm is famous [57] for finding approximations with $|\alpha - p/q| \leq 1/2q^2$. This spurred the field of number theory to study generalizations of the continued fraction algorithm that come close to Dirichlet's upper bound, but avoid brute-force calculations..

2.5.4 Reductions of Spherical Predicates to Cartesian Orientation Predicates

During the construction of a Delaunay triangulation in $\mathbb{R}^2$ one often has to answer the question whether a point q lies left/on/right of the ray r starting in $p1$ and passing through $p2$. We call an oracle able to answer such a question a predicate. In higher dimensions we can formulate it as whether the point $q\mathbb{R}^n$ is "left"/on/"right" of the hyperplane P. On the sphere this translates to the questions whether the point q is left/on/right of the great circle defined by the two points $p1$ and $p2$.

Lemma 1 (Great Circle Orientation Predicate). *Let $p_1, p_2 \in \mathbb{S}^2$ with $p_1 \neq p_2$ and P the plane containing p_1, p_2 and the origin $(0,0,0)$ and C be the Great Circle*

through p_1 and p_2. For $q \in \mathbb{S}^2$ we have

$$\begin{aligned} q \text{ left-of } P &\iff q \text{ left-of } C \\ q \in P &\iff q \in C \\ q \text{ right-of } P &\iff q \text{ right-of } C \end{aligned}$$

Another used predicate is the in-circle predicate which tells whether a point $q \in \mathbb{R}^2$ is inside/on/outside a circle defined by three points $p_i \in \mathbb{R}^2, i = 1 \dots 3$. The same questions arises while computing a Delaunay triangulation on the sphere which we can also answer using a simple point-plane orientation predicate.

Lemma 2 (Circumsphere Predicate)**.** *Let P denote the plane through non-identical points $p_1, p_2, p_3 \in \mathbb{S}^2$ and the half space containing the origin $(0, 0, 0)$ is called 'below P'. We further call $S_{123} \subseteq \mathbb{R}^3$ the closed volume of the sphere with p_1, p_2, p_3 and the origin on its surface. For a point $q \in \mathbb{S}^2$ we have*

$$\begin{aligned} q \text{ above } P &\iff q \in S_{123} \setminus \partial S_{123} \\ q \in P &\iff q \in \partial S_{123} \\ q \text{ below } P &\iff q \notin S_{123} \end{aligned}$$

See [16] for a proof.

2.6 Miscellaneous

2.6.1 Cumulative Distribution Function

We are extensively using cumulative distribution functions in our experimental evaluation. For a one dimensional random variable X with probability P its cumulative distribution function (cdf) is $F_X(x) = P(X \leq x)$. In our case we often have as input a set $\mathcal{S}$ of strings and for each string a measurements m like the time it took to process the string. We are interested in the cdf of $P(m) = \frac{1}{|\mathcal{S}|}$ which gives us the percentage of strings that can be processed in less than a given time M. See figure 2.9 for an example.

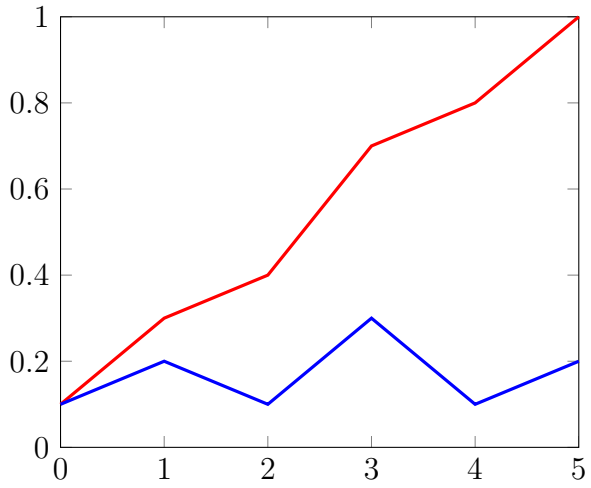


Figure 2.9: Blue: Probability distribution P of a random variable $X \in [0 \dots 5]$ and its cumulative distribution function $F_X(x) = P(X \leq x)$ in red.

3 The OSCAR Search Engine

In the following sections we would like to introduce the building blocks of OSCAR. We start by giving a definition of the cell arrangement that is at the heart of OSCAR's processing capabilities. We then introduce the various context aware spatial relation operators. Finally we show how these can be answered using the cell arrangement and how text and spatial queries can be used together for complex textual-spatial queries.

3.1 Map Tessellations and Cell Arrangements [4]

Maps of geospatial data are often stored and accessed through tessellations of the input data. Most notably regular tessellations using rectangles are used to access the data. An example of such a tessellation of the Baden-Württemberg data set from the OpenStreetMap projected is depicted in figure 3.1. Our data structure used to solve the problems defined in 3.3 is based on a specially crafted tiling which we will define in the following section.

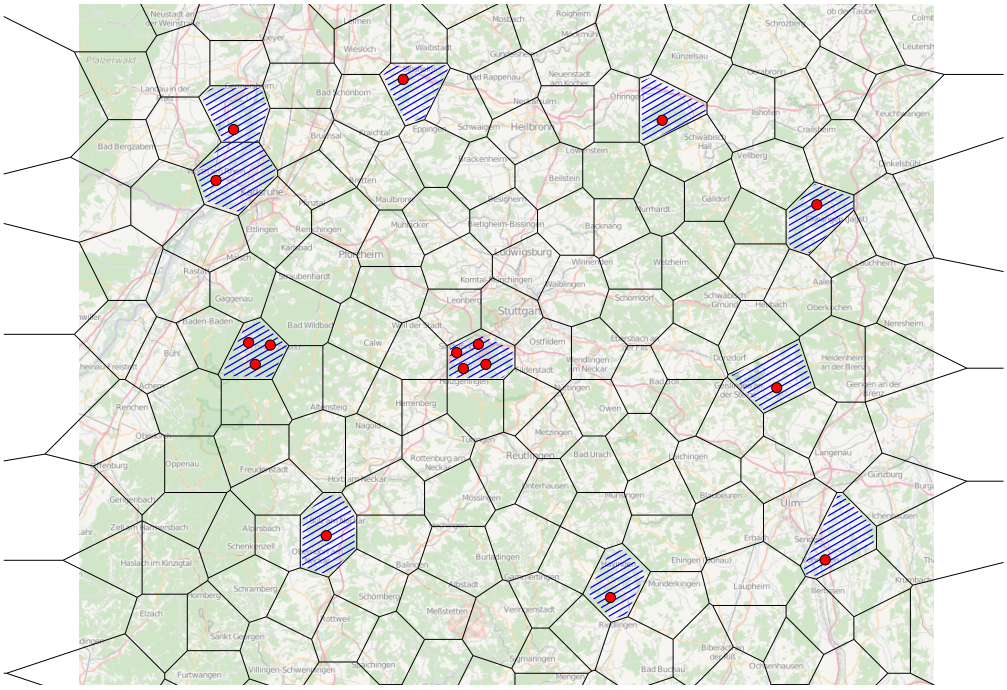


Figure 3.1: Decomposition of a map into tiles with some tiles marked in blue containing items represented by red dots.[5]

3.2 OSM Cell Arrangement [6]

The key concept behind OSCAR is a so-called *cell arrangement of OSM regions*. Consider a set of polygons $\mathcal{R}$, each corresponding to an OSM region. Each single polygon $r \in \mathcal{R}$ divides the plane into two regions, the *interior* of the polygon and the *exterior*. The set $\mathcal{R}$ naturally induces a subdivision of the plane into *cells*, see Figure 3.2, for an example, we call this the *cell arrangement* $\mathcal{C} = \mathcal{C}(\mathcal{R})$ of $\mathcal{R}$. All points contained in a single cell $c \in \mathcal{C}$ have the property that they behave identically with respect to containment in the set of polygons, in particular inheriting the same information associated with respective OSM regions. Note that a region may consist of multiple simple polygons describing the outer boundary together with a set of simple polygons describing inner boundaries thus introducing holes. A consequence of this is that cells are not connected, may have holes filled with other cells and be of different size on a broad scale. If textual search is our only interest then this does not pose any problem. However using the cell arrangement to implement spatial queries is a difficult task. Consider for example a query where we want to get all the cells intersecting a polygonal chain. If we could navigate the cell arrangement then we could simply walk along the segments and

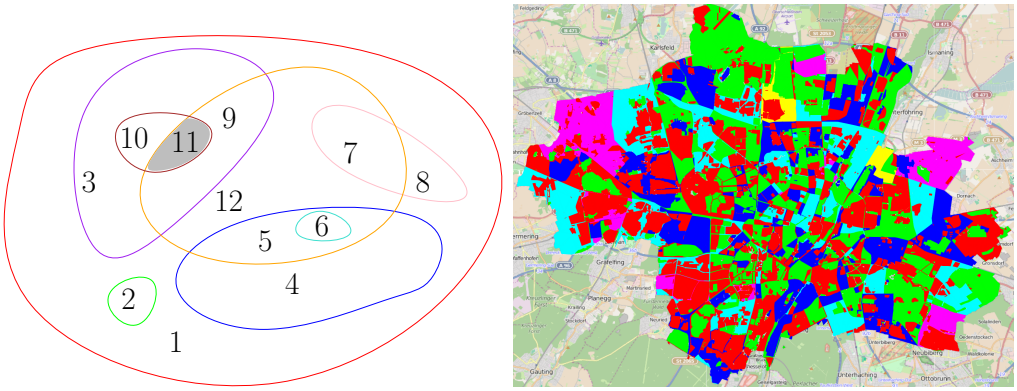


Figure 3.2: Left: A cell arrangement induced by 8 regions creating 12 different cells. Right: A cell arrangement of the regions in Munich. Neighboring cells have different colors. There are about 800 cells in Munich.

gather all the cells that we encounter. We therefore refine the cell arrangement to get nicer cells that are easier to work with. To this end we compute a constrained Delaunay triangulation of the input regions. See section 2.3.1 for the properties of this special type of triangulation. A cell then covers a set of triangles which we use to represent its geometry. A simple way to refine a cell is to partition the set of its reference triangles such that a given quality measure is fulfilled. Depending on the triangulation it may however not be possible to fulfill the quality measure – consider for example a measure setting a maximum diameter of the cell and a triangle that is larger than this threshold. We therefore also refine the triangulation depending on the chosen quality measure. See figure 3.3 for a conceptual illustration. Figure 3.4 on the other hand depicts the difference of an unrefined and a refined cell near the village Alfdorf in Germany. We will cover the details in the following section.

Refined Cell Arrangement

In order to compute the refined arrangement we can use a top-down approach where we split cells or a bottom up approach where we define the triangles to be cells and merge cells to form new cells. We use the splitting approach and start by splitting each cell into its connected components and continue splitting the connected components until the given quality measure is fulfilled. To this end consider the graph G_N induced by the neighborhood relation of the triangles of the underlying triangulation of the cell. In order to split this graph into k sub-graphs we first find two nodes, the generators, that are furthest apart from each other with respect to hop-distance in the graph. These nodes define the diameter of the graph

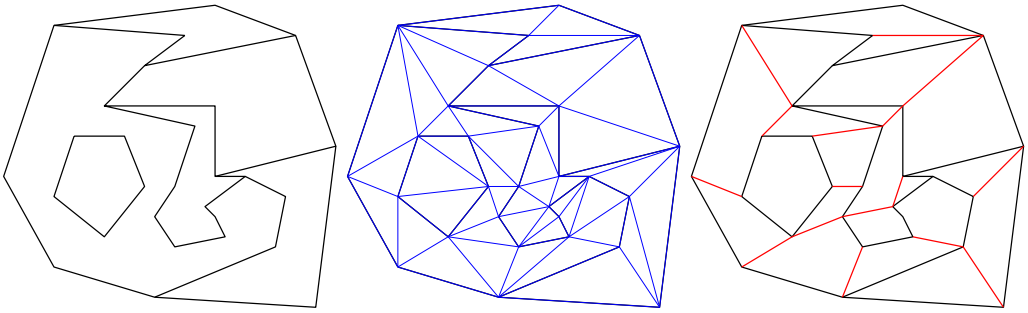


Figure 3.3: Schematic of the refinement process. Left: Original Cell Arrangement. Center: Constrained Delaunay Triangulation thereof. Right: Sets of triangles forming the refined cell arrangement where red edges were not part of the original arrangement.[7]

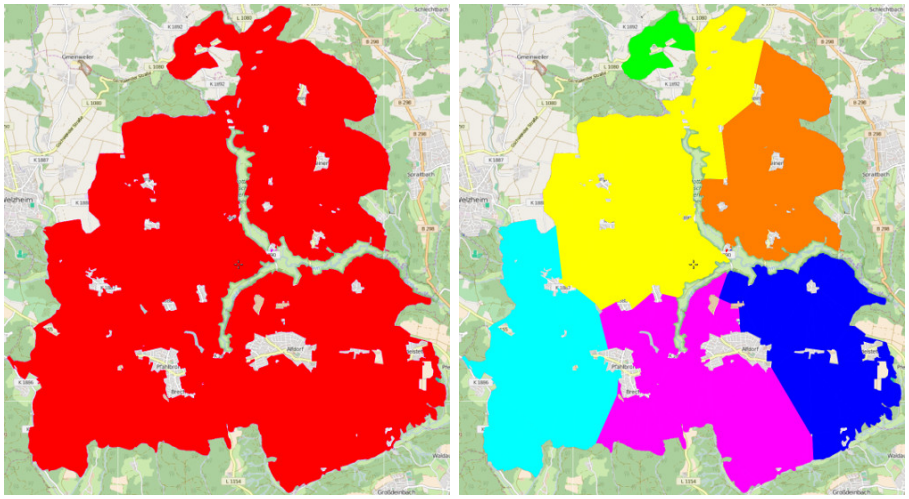


Figure 3.4: A cell before and after refinement.[8]

and are representatives for the two new cells. We compute these nodes exactly for small graphs and approximately for large graphs. We then assign each node of the graph the cell id of the closest generator with respect to hop distance. We add more generators that are furthest apart from every other generator as long as the cell is below the defined quality threshold. In the following we would like to give a short description of possible measures to produce the refined cells.

Refine by Size of the Cell The first measure that comes to mind is to split a cell as long as its diameter is larger than a given threshold. This way one can influence the approximation quality of all spatial operators that rely on the cell's geometric size, most notably the neighborhood query. Using 1.5 kilometers, about 1 square kilometer, as an upper bound would result in a very good approximation but also produce about 150 Million cells with many cells containing no items at all.

Refine by Item Density Refining cells by the number of items covered may also be helpful for the approximate neighborhood query. The neighborhood in a densely populated region is often subjectively smaller than in a sparsely populated area. Consider for example the locations of supermarkets of which there are many more in cities than on the countryside and thus reducing the distance needed to travel to buy food. Hence a user living in a city has a different notion of "nearby" than someone living on the countryside.

Refine by Number of Triangles If the size distribution of the triangles is more or less homogeneous one can easily compute well-shaped cells by only considering the number of triangles the cell is made up of. We furthermore observed that densely populated areas are covered by more unrefined cells as well as smaller triangles. Considering the aforementioned observation and depending on the intended semantic of the neighborhood query it may even be counter productive to ensure homogeneous size distributions. Instead if we refine the cells according to the number of triangles results in smaller cells in densely populated areas and larger cells in sparsely populated areas.

Triangulation Refinement Some cell refinement criteria need to have a refined triangulation as well. Consider for example the cell diagonal criterion which cannot be fulfilled if the triangles are too large. Hence we also refine the triangulation beforehand. There are many options to create "nice" triangulations. We can compute a conforming Delaunay triangulation by adding vertexes such that every constrained edge is also a Delaunay edge – hence a Delaunay triangulation. We get a conforming Gabriel triangulation if we impose the stronger condition that for each edge the smallest enclosing circle shall not contain any vertexes of the

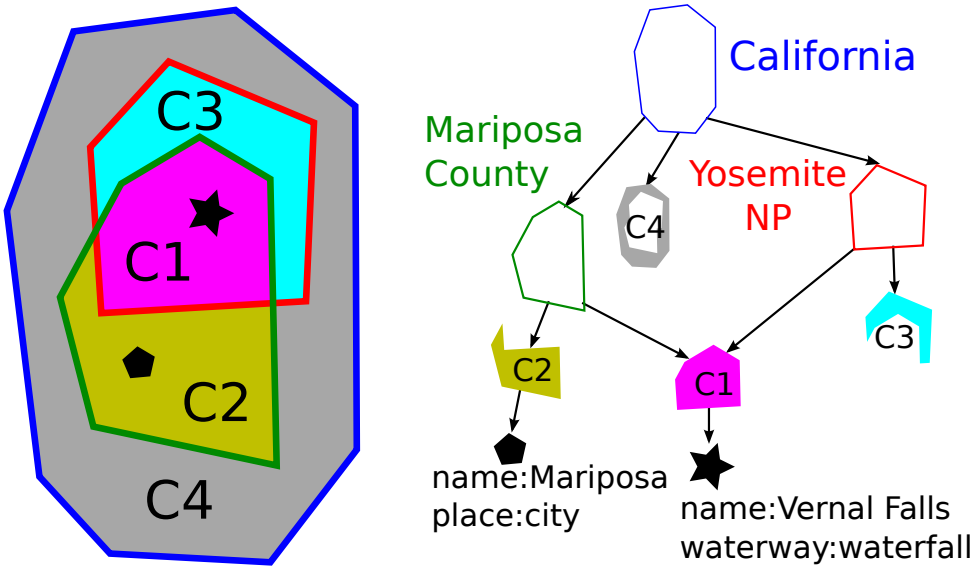


Figure 3.5: The hierarchy induced by the three regions "California", "Mariposa County" and "Yosemite National Park" forming the cells C1 through C4 with items "Mariposa city" and "Vernal Falls"

triangulation. Other quality measures may be based on the maximum edge length, edge-length ratio or diameter of the circumscribed circle of a triangle. Refinement may also be based on the neighborhood of a triangle such that a given quality measure does not dramatically change in the vicinity of a triangle.

3.2.1 Inclusion-DAG

The inclusion-relation of the regions $\mathcal{R}$ defined in the data set induces a directed acyclic graph which we refer to as the Inclusion-DAG or inclusion hierarchy. Each region is represented by a node. There exists an edge from node p to c if and only if the region of c is covered by p and there is no other region that is covered by p that also covers c. We furthermore add the cells of the arrangement to the hierarchy by introducing a cell node for each cell. There is an edge from a region to a cell node if and only if the region covers that cell. Finally all items are also added as item nodes to the hierarchy with edges from cell nodes to item nodes if the cell has a non-empty intersection with the item. See figure 3.5 for a conceptual visualization.

3.3 Semantics of Spatial Relations [9]

The cell arrangement allows for an easy implementation of various spatial relation operators. To be precise we are interested in the following spatial relations:

- neighborhood (*near X*)
- cardinal directions (*north-of X*, *south-of X*,...)
- betweenness (*between X and Y*)
- path corridors (*along route X*)

However the first 3 relations depend on the context, namely on the object X. Clearly the region defined by north-of "Germany" should be different in size compared to north-of "Hauptbahnhof Stuttgart". In the following we would like to give context sensitive definitions for these spatial relation operators.

3.3.1 Neighborhood

The probably most natural spatial relation is about proximity, but there are considerable differences in terms of semantics depending on the objects referenced. For example, when asking for a 'restaurant near the Eiffel Tower', one would probably refer to locations at most 500m away from the Eiffel Tower. On the other hand, a 'hotel near Paris' might refer to locations up to 20km away from the city of Paris, 'a motel near the Rocky Mountains' to an area with an extent of several thousand kilometers.

In the above examples we used common sense to determine up to what distance we are interested in matches (500m in case of the Eiffel Tower, $\approx$ 20km in case of the city of Paris). How could a search engine simulate this common sense? The extent of the reference object (the Eiffel Tower or the city of Paris) typically also determines the extent of the region of interest. To keep things simple, we compute a minimum enclosing rectangle for the reference object and let d be the extent of the *smaller* dimension (breadth or width). Then we define as neighborhood of a reference object all locations which have distance at most $\max(d, 500m)$ to the reference object. We use the maximum of d and 500m to also define neighborhoods of 0- or 1-dimensional entities (e.g., a mailbox is typically not considered to have any real 'extent'). See Figure 3.6 for an example.

3.3.2 Cardinal Directions

Things are quite similar for cardinal directions. What would a user expect when issuing a query like "*hotels north of the Empire State Building*"? Probably accommodations within a distance of 500m north of the Empire State Building. For

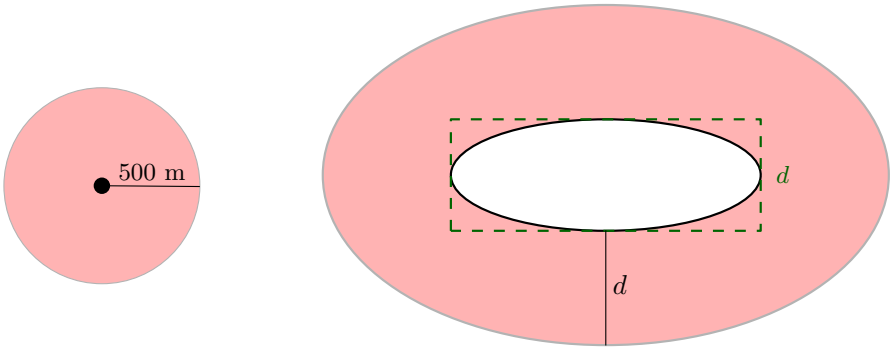


Figure 3.6: Examples for neighborhood determination of a 0-dimensional and a 2-dimensional object.

larger entities, a larger region is of interest. For example, asking for "*hotels north of Paris*" probably refers to a region up to a distance of 20km north of Paris.

Trying to formalize this common sense, we determine the extent of the region of interest as follows, restricting to the "north-of" relation (north-east, east, …work likewise).

For 0-dimensional reference objects a simple triangle in the cardinal direction is used. In fact in this case we cannot infer the size of the region of interest from the extent of the reference object and use a triangle of height 500m. See Figure 3.7. If additional context information is available, the size of this triangle can easily be adjusted.

For 1-dimensional reference objects, we first determine the northernmost point of the polygonal line as well as the midpoint of the bounding box. We then use the point cardinal direction function to create an intermediate triangle based on the northernmost point and the mid point of height twice the width of the bounding box. The region of interest is then the convex hull of this triangle together with the polygonal line. See Figure 3.7. For 2-dimensional reference objects we first create a bounding box and an isosceles trapezoid of height twice the height of the bounding box, base equal to the horizontal axis of symmetry of the bounding box and a parallel opposing side of double the length of the base. See Figure 3.7 for a schematic drawing.

3.3.3 Betweenness

For a query "*hotels between Frankfurt and Cologne*" we would expect hotels in a corridor of maybe 10–20km width between the German cities of Frankfurt and Cologne (around 170km apart). On the other hand, a query "*restaurants between Golden Gate Bridge and Moscone Center*" in San Francisco should have the re-

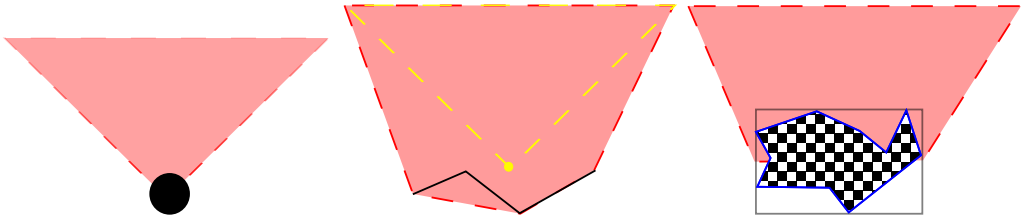


Figure 3.7: Regions of interest for 0-, 1- and 2-dimensional reference objects (red, dashed) for the 'north of' relation. The yellow triangle is the intermediate triangle used to create the polygon.

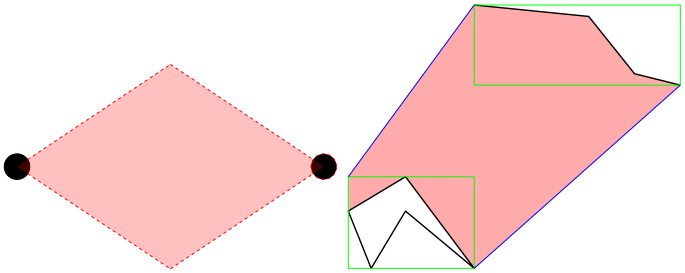


Figure 3.8: Regions of interest for 0-, 1- and 2-dimensional reference objects (red, dashed) for the 'between' relation.

sults restricted to a much smaller area of few square kilometers. As for cardinal directions, the region referred to by a 'between' relation depends on the reference regions and in particular their distance from each other.

To actually compute polygons bounding the region of interest for the betweenness relation we again distinguish between the dimension of the reference objects. If we are after the region between two 0-dimensional reference objects, we construct a diamond shape with the corners being the two reference objects. In all other cases we construct the convex hull of the bounding boxes of both reference objects. We use the area of the convex hull without the bounding boxes as region of interest, as well the part of the bounding boxes that is formed by connecting the convex hull edges between the bounding boxes to the next nodes of the reference object on the bounding box border in the respective direction (which must be well-defined). See Figure 3.8 for an illustration.

3.3.4 Path Corridors

When looking for 'hotels along the route from Stuttgart to Berlin' one typically refers to a much smaller area than in 'hotels between Stuttgart and Berlin'. So

given a (precomputed) path we define as the area of interest as all locations with distance of at most 1 kilometer from the route.

We want to note, though, that for all discussed relations other toolboxes than the ones described here (or with different parameter choices) could be plugged in our system as well.

3.4 Textual Spatial Queries Based on OSM Cell Arrangements

In the following section we describe how text matching, geometric matching and complex spatial relations can be implemented based on the aforementioned cell arrangement.

3.4.1 Result Representation

The result of every query statement are two sets: The set of full-match cells and the set of partial-match cells. A full-match cell is a cell where all covered items are a match for the query. The partial-match cells on the other hand have at least one item, but not all, matching the query. For reasons of simplicity we may assume that each cell in the partial-match cells set also explicitly stores the matching items. Additionally each cell and item has a unique identifier $i \in \mathbb{N}$. In short we have:

$$\mathcal{C}_f(q) \subset \mathbb{N}$$
$$\mathcal{C}_p(q) \subset \mathbb{N} \times \mathcal{P}(\mathbb{N})$$

3.4.2 Set Operations

Set operations such as intersection, union, difference and symmetric-difference can be computed based on the full-match and partial-match cells. For an intersection it can be computed as follows:

$$\begin{aligned}
\mathcal{C}_f(q_1 \sqcap q_2) &= \mathcal{C}_f(q_1) \cap \mathcal{C}_f(q_2) \\
\mathcal{C}_p(q_1 \sqcap q_2) &= (\mathcal{C}_p(q_1) \cap \mathcal{C}_f(q_2)) \cup (\mathcal{C}_p(q_2) \cap \mathcal{C}_f(q_1)) \cup (\mathcal{C}_p(q_1) \cap \mathcal{C}_p(q_2)) \\
&= \{(c, I) \in \mathcal{C}_p(q_1) : c \in \mathcal{C}_f(q_2)\} \\
&\cup \{(c, I) \in \mathcal{C}_p(q_2) : c \in \mathcal{C}_f(q_1)\} \\
&\cup \{(c, I_1 \cup I_2) \in \mathbb{N} \times \mathcal{P}(\mathbb{N}) : (c, I_1) \in \mathcal{C}_p(q_1) \wedge (c, I_2) \in \mathcal{C}_p(q_2)\}
\end{aligned}$$

All other set operations can be computed analogously. See figure 3.9 for a graphical visualization.

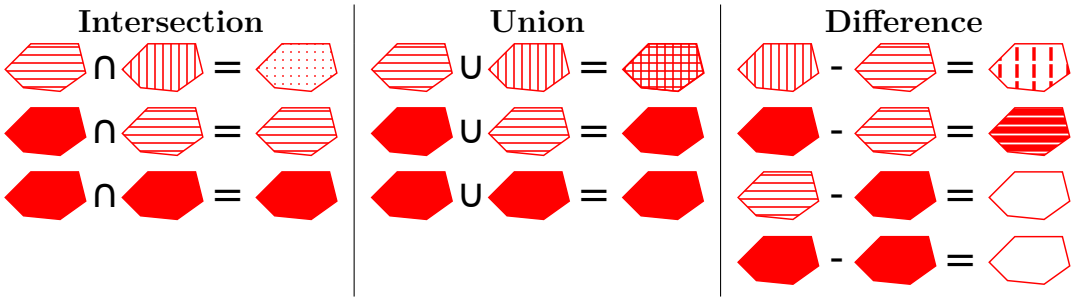


Figure 3.9: Graphical visualization of set operations based on cells. Stripped or dotted cells are partial-match cells whereas filled cells are full-match cells.

3.4.3 Text Matching

For the text matching we distinguish between items and regions. Querying for items with a text query q returns the smallest set of cells covering all matching items. Querying for regions on the other hand returns the set of cells that are covered by all regions matching the query. By default OSCAR returns the union of these queries and only returns the item or region query result on request by the user.

3.4.4 Polygon Intersection

Retrieving the cells or items intersecting a given polygon can be answered with the Inclusion-DAG, the cell arrangement or the triangulation. Polygon queries that intersect only a small amount of cells can be answered by locating the polygon in the cell arrangement and explicitly checking nearby cells. For larger polygons the Inclusion-DAG can be interpreted as an R-tree to retrieve all intersecting cells. The accuracy of the polygon intersection test can be varied from exact to very approximate. For an exact result a post-processing step is necessary to only include items intersecting the polygon. This step can either be done using the exact geometry of the items, their bounding box, the exact geometry of the polygon or its bounding box. The same can be done analogously for polygon-cell and polygon-region intersections. Hence the exact extreme is the intersection of the geometry of items and the polygon. On the other end is the intersection of the bounding box of the polygon with the intersection of the bounding box of regions while traversing the Inclusion-DAG.

3.4.5 Along Path

Computing the cells along a given path can easily be achieved by traversing the triangulation of the cell arrangement and marking all encountered cells. See figure 3.10. This could be improved if the path is constructed from a shortest-path query: Store for each edge all intersecting cells and collect these cells from the given path. Further improvements can be made by using contraction hierarchies [45] where an edge may represent a longer chain of edges and hence stores the union of their cells.

3.4.6 Nearby

The simplest method to compute the *nearby* relation is to construct a polygon around a given object and pass it to the polygon intersection operator. However this is only efficient for a single object. Computing the *approximate* nearby relation of a large set of objects can be achieved based on the cell arrangement. For a given set of cells we simply mark all nearby cells as containing nearby elements. In image manipulation this operation is called *dilation* — henceforth we will refer to this operation as a dilation operation as well. The dilation operator may expect an additional parameter in the form of a distance to decide if a cell is near the input cell set. See figure 3.11 for a illustration.

3.4.7 Cardinal Direction

The cardinal direction operator can be computed in two ways as well. We can either use the polygon defined in 3.3.2 or use a cell dilation operator with a cardinal direction parameter. To be precise we are given a set of input cells, a distance parameter k and a cardinal direction parameter d. We now mark all cells that have a distance of at most k and a cardinal direction d to at least one cell of the input set (see figure 3.12)

3.4.8 Betweenness

Answering a between operator is based on the polygon outlined in section 3.3.3. This polygon is then passed to the polygon intersection operator. A schematic visualization is shown in figure 3.13.

3.4.9 Putting It All Together

All operations mentioned before can be used together in a single query. Some query statements need additional information to work. These are the Cardinal direction operator and the betweenness operator that both need a reference object to start with. We extract these objects from the sub-queries that are inputs to the

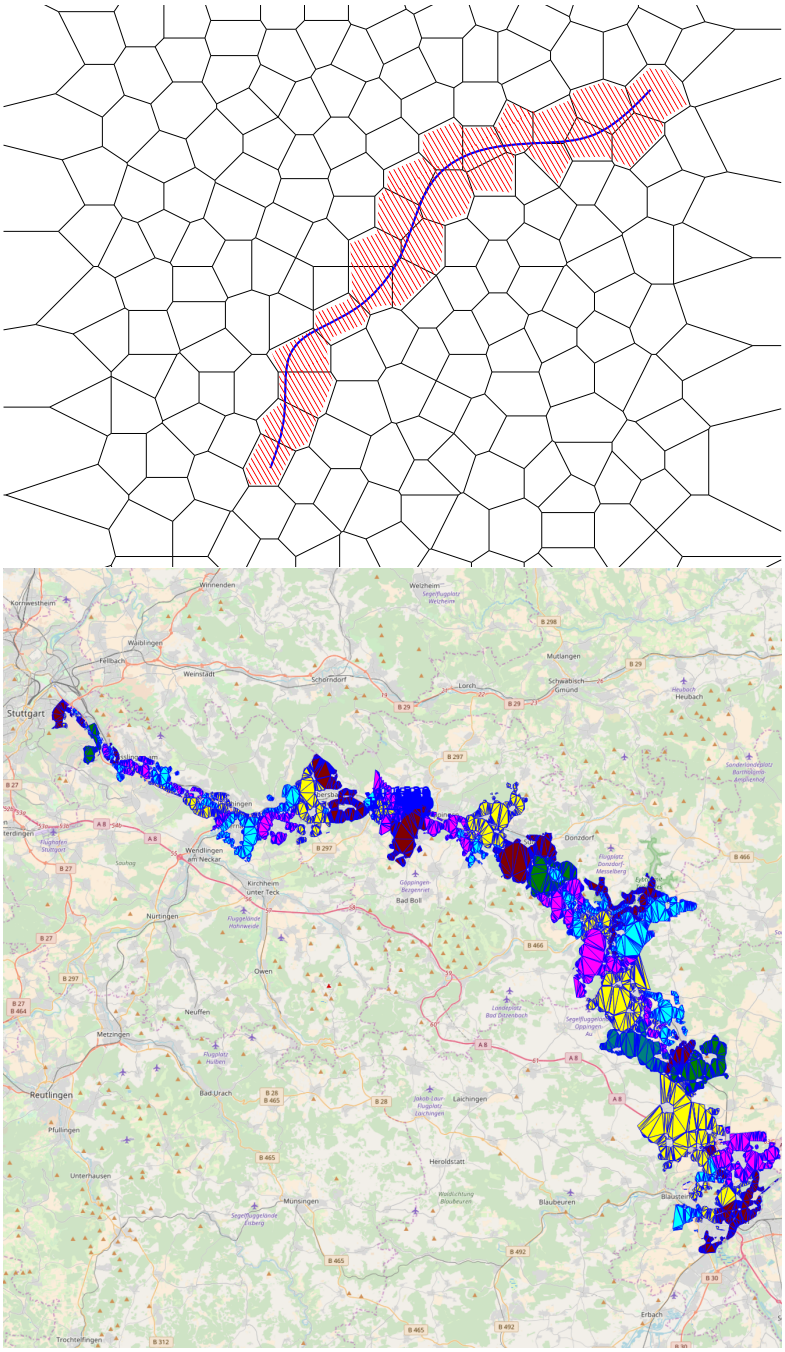


Figure 3.10: Top: Marked cells of a *along-path* query.[10] Bottom: Cells intersected by a path from Stuttgart to Ulm along the B10.

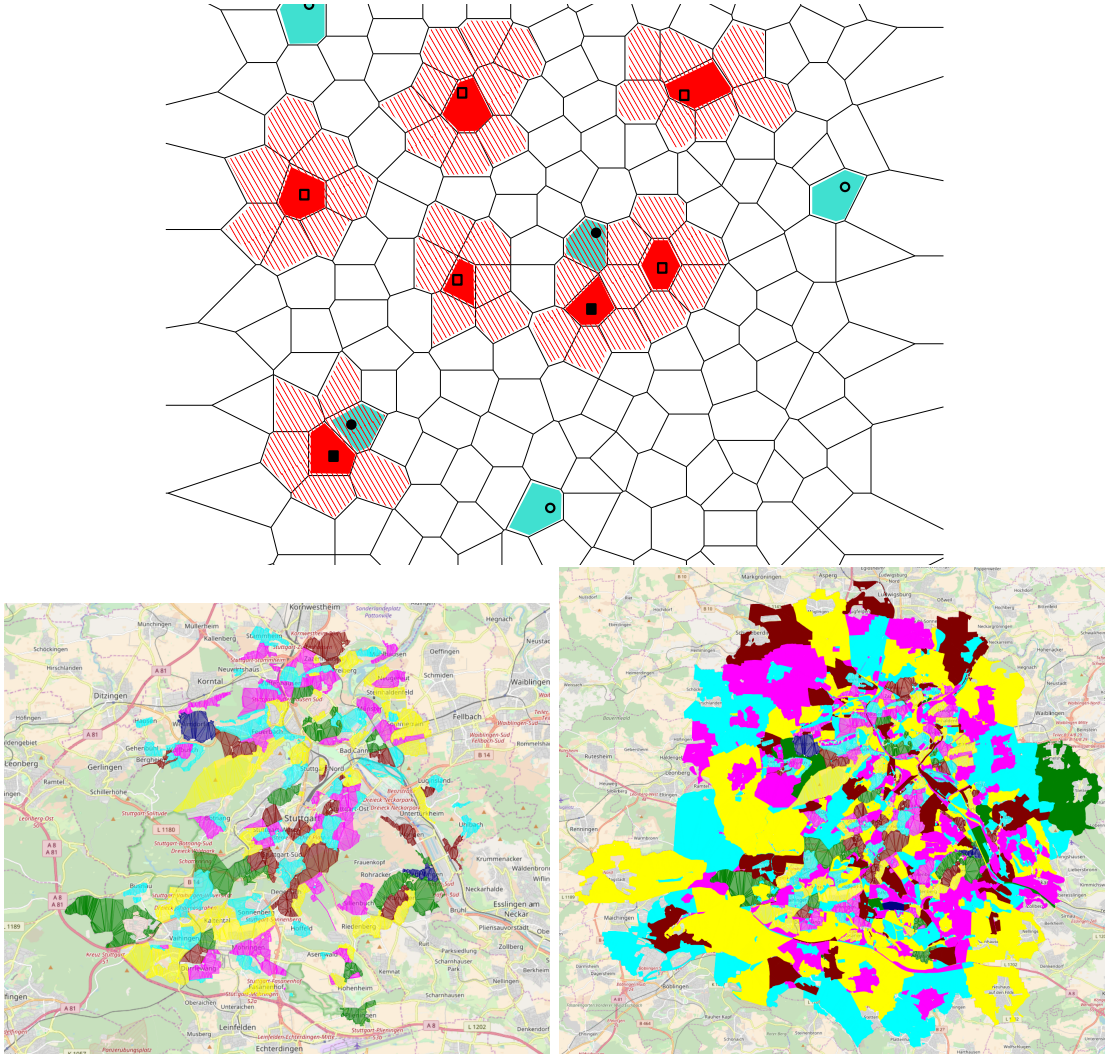


Figure 3.11: Top: Marked cells of two (red, green) *Nearby* queries using a ranged cell dilation operator.[11] Mid: Cells of the query "*@amenity:kindergarten #"Stuttgart""*. Bottom: Cells of the query "*%2% (@amenity:kindergarten #"Stuttgart")*" dilating the former query by 2 kilometers.

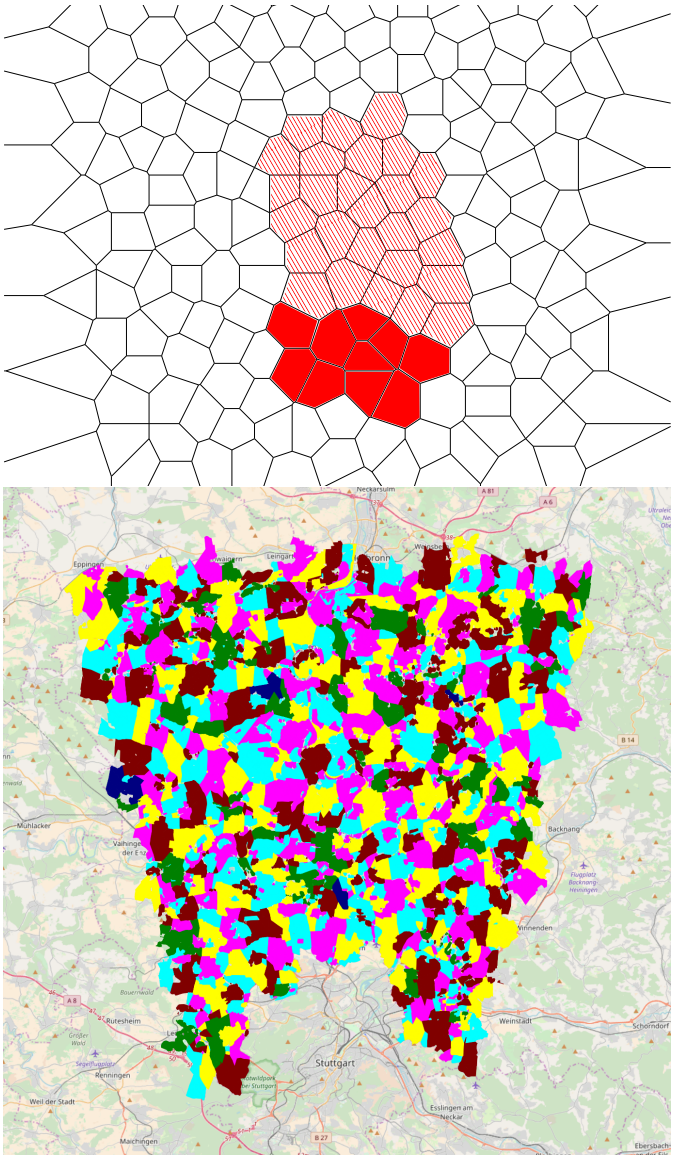


Figure 3.12: Top: Marked cells of a *NorthOf* query.[12] Bottom: Cells of the query "*:north-of #"Stuttgart""*.

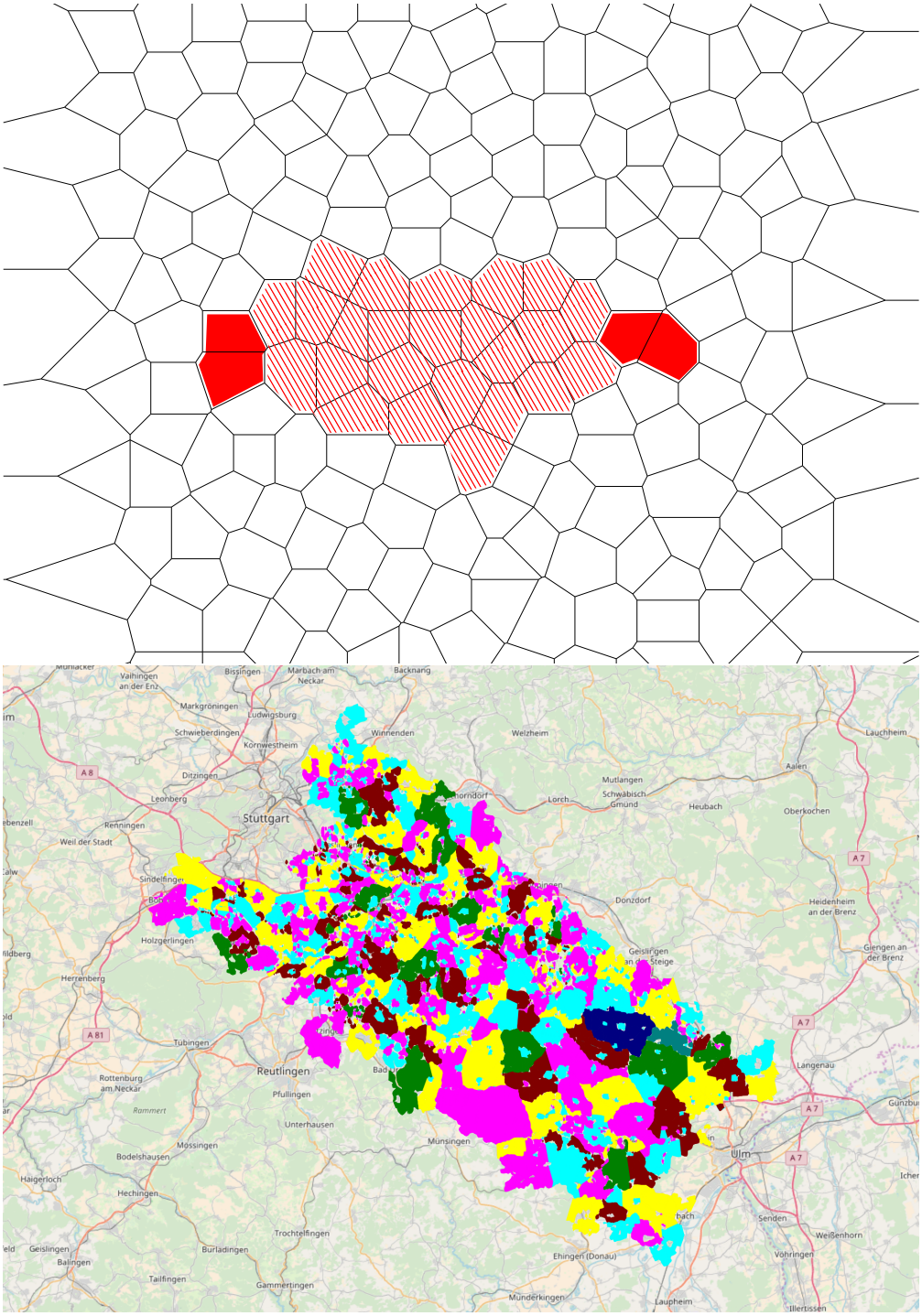
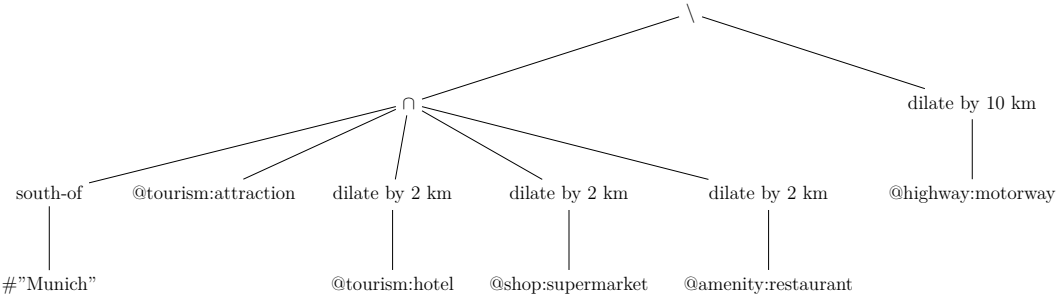


Figure 3.13: Top: Schematic result of cells marked of a *Between* query.[13] Bottom: Cells of the query "*#"Stuttgart" <-> #"Ulm""*.

spatial operators. Let us consider the query mentioned in the introduction which OSCAR may parse into the following operation tree:



We can compute the result of these operations using the previously introduced search operations as follows: We first compute the result of the text queries which is a simple look-up in our text search data structure. We construct an appropriate polygon for the "*south-of #"Munich""* query and use the polygon intersection query to retrieve the matching cells. We use the dilation operation to retrieve the cells for the 4 dilation queries. The intersection operation returns all the tourist attractions south of Munich that have a hotel, supermarket and restaurant nearby. We then remove all those that have a highway up to 10 km away by dilating all matching highways by 10 km and removing all those cells from our previous result. The result then only contains those cells that are more than 10 km away from a highway and contains at least one tourist attraction near a hotel, supermarket and restaurant.

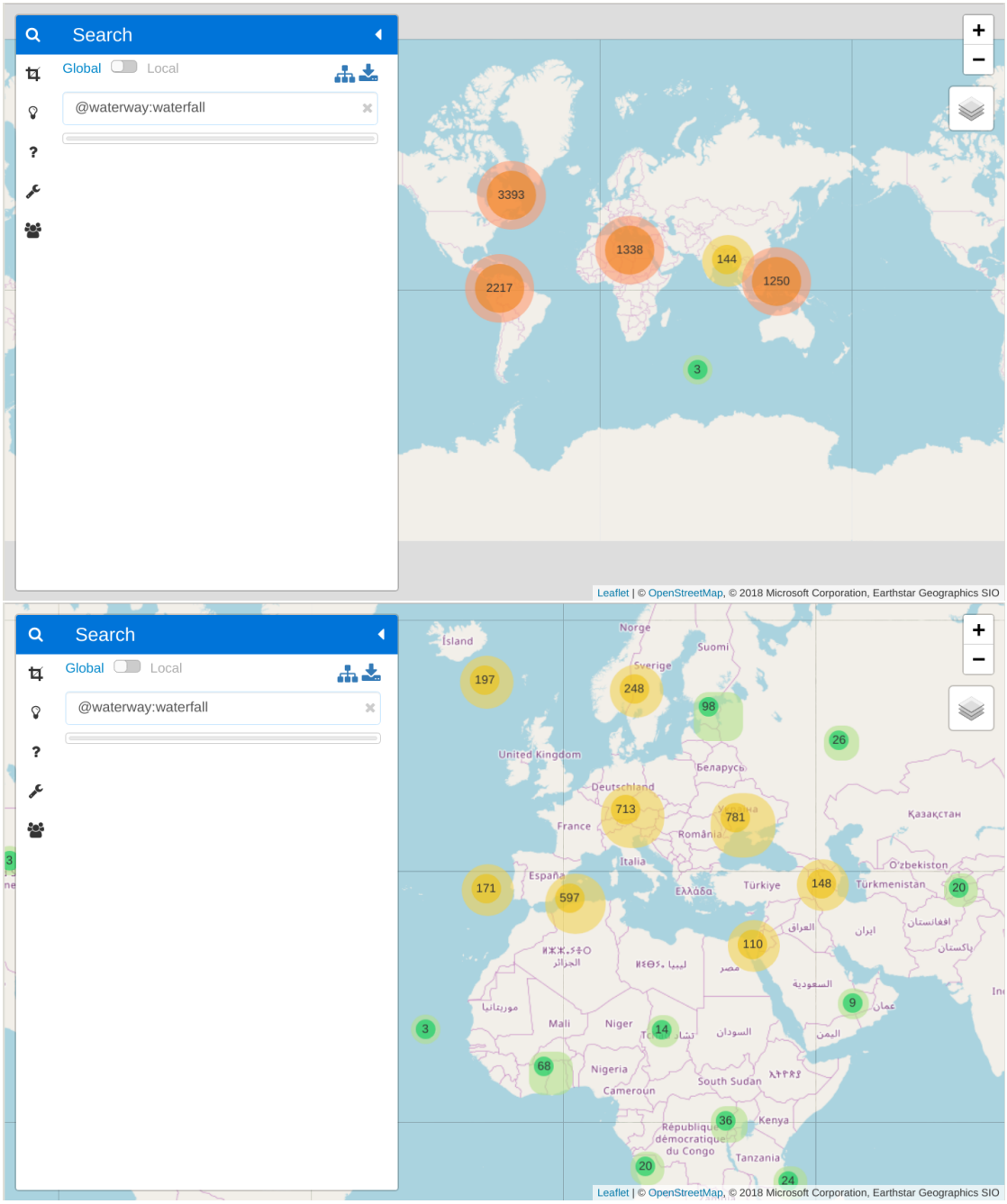
3.5 Theoretical Performance

OSCAR relies on the fact that there are more cells than items and hence cells contain a large number of items. The more cells there are compared to the number of items, the longer a computation of a query result takes. If we only consider point-like items then the we have that there cannot be more cells than items since we can simply omit cells containing no items. Hence computing a result for a query where every cell contains only a single point-like item should result in a performance within a constant factor of a simple inverted index. Note that items consisting of lines or polygons may span multiple cells. In the worst-case a single item may span all cells and thus produce results containing all cells compared to just the single item. However in practice this is of no concern since items only span a very small number of cells.

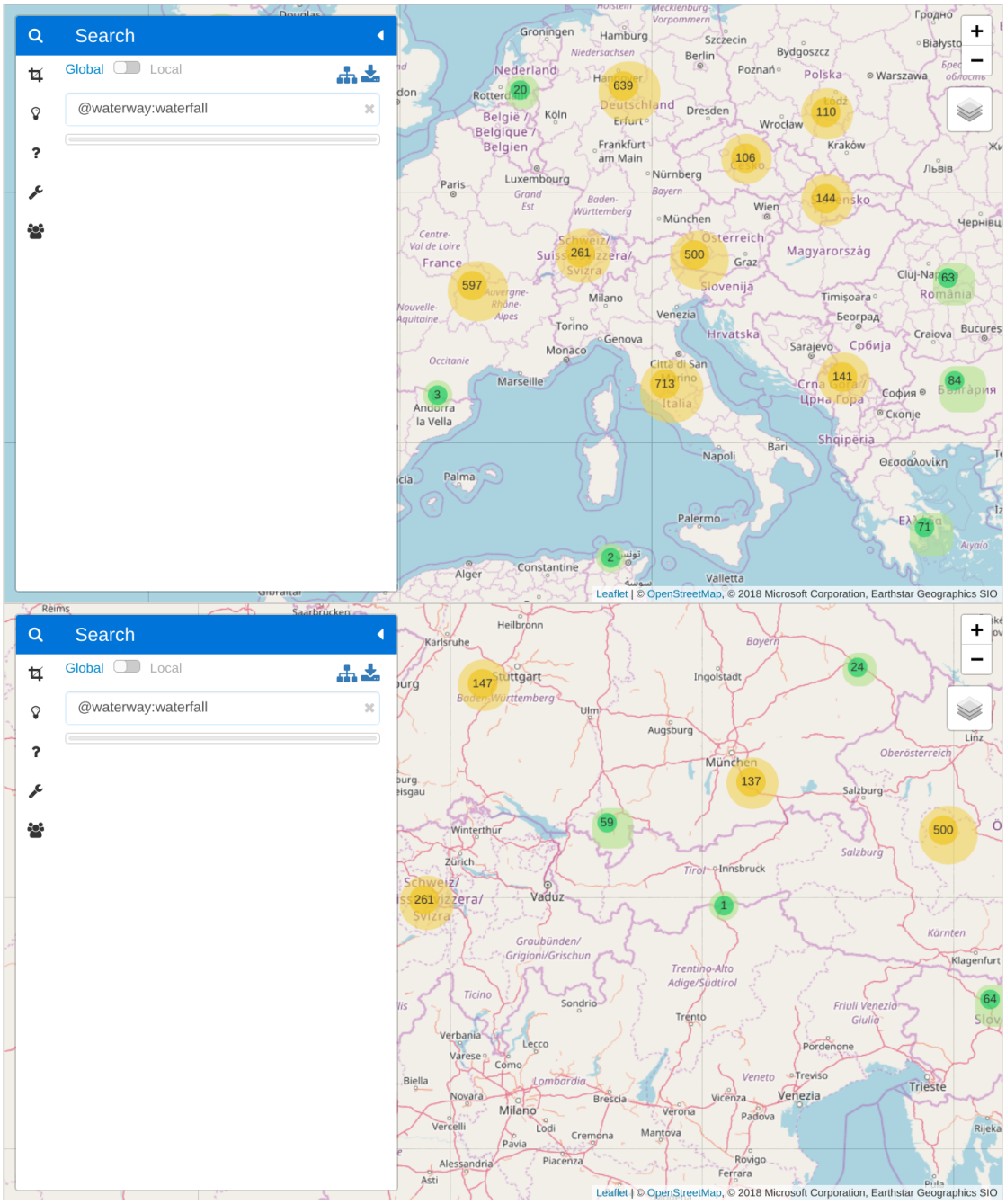
3.6 Result Presentation

A natural presentation of a query result is the subgraph of the Inclusion-DAG defined by all regions, cells and items that are either part of the result or have a descendant that is part of the result. This graph can also be used to present aggregated information like the number of items in a particular region. The subgraph can be computed based on the full-match and partial-match cells obtained from the query. For each cell in the result we simply mark each ancestor and cumulate the number of items of the cell. Note that this is different from a simple aggregation using clustering algorithms or quad-trees in that the regions can be seen as a form of semantic clustering providing the user meaningful aggregate information. We have implemented a web based demonstrator to explore the usability of this approach. Higher zoom levels use the Inclusion-DAG to draw cluster icons with approximate item counts. On lower zoom levels the items are drawn using a marker icon. Clicking on an item draws its spatial data and presents its associated tags. In the following we would like to give the reader a glimpse of OSCAR's visualization based on the query "*@waterway:waterfall*".

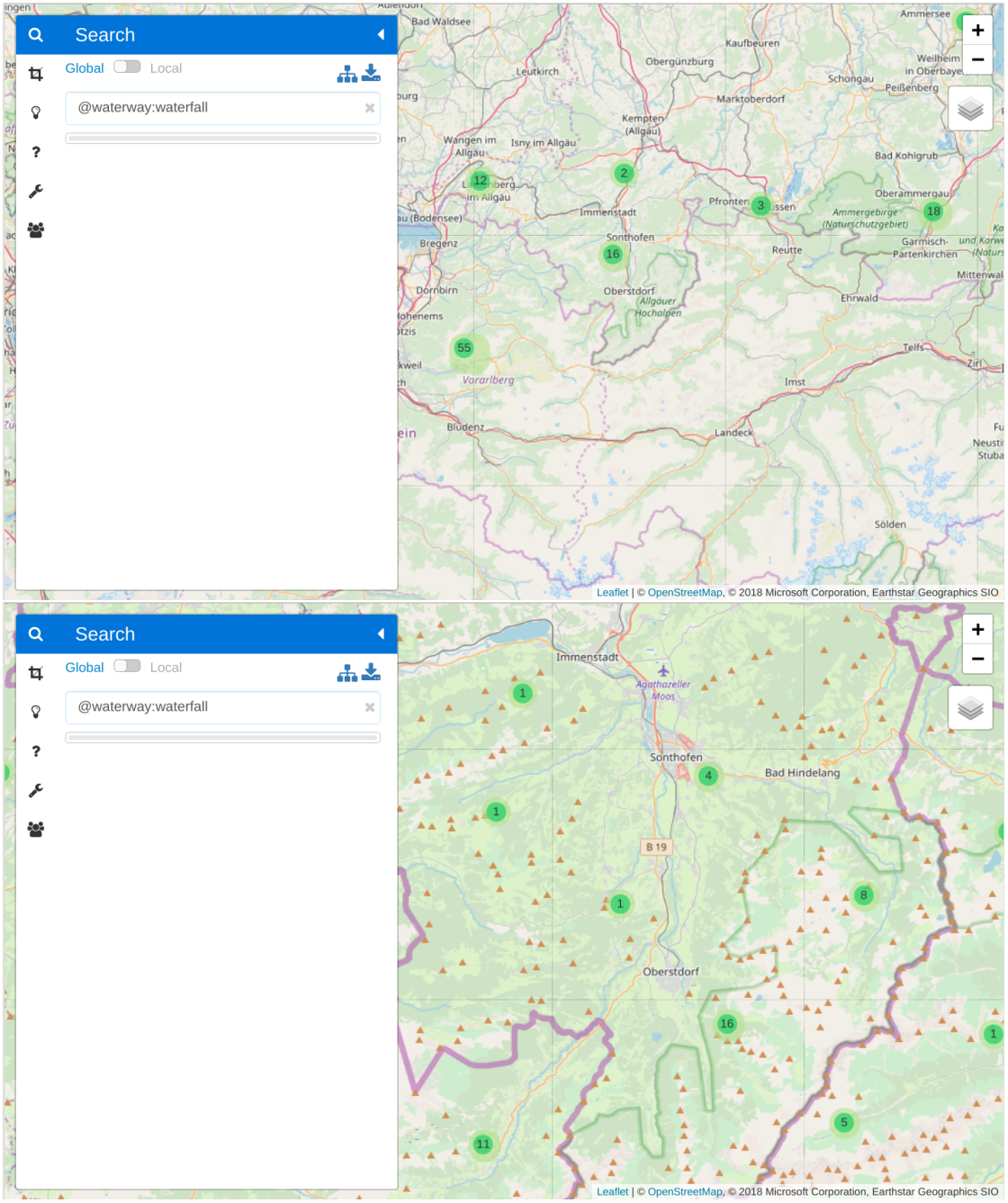
We begin with an overview of the world with all waterfalls of the data set:



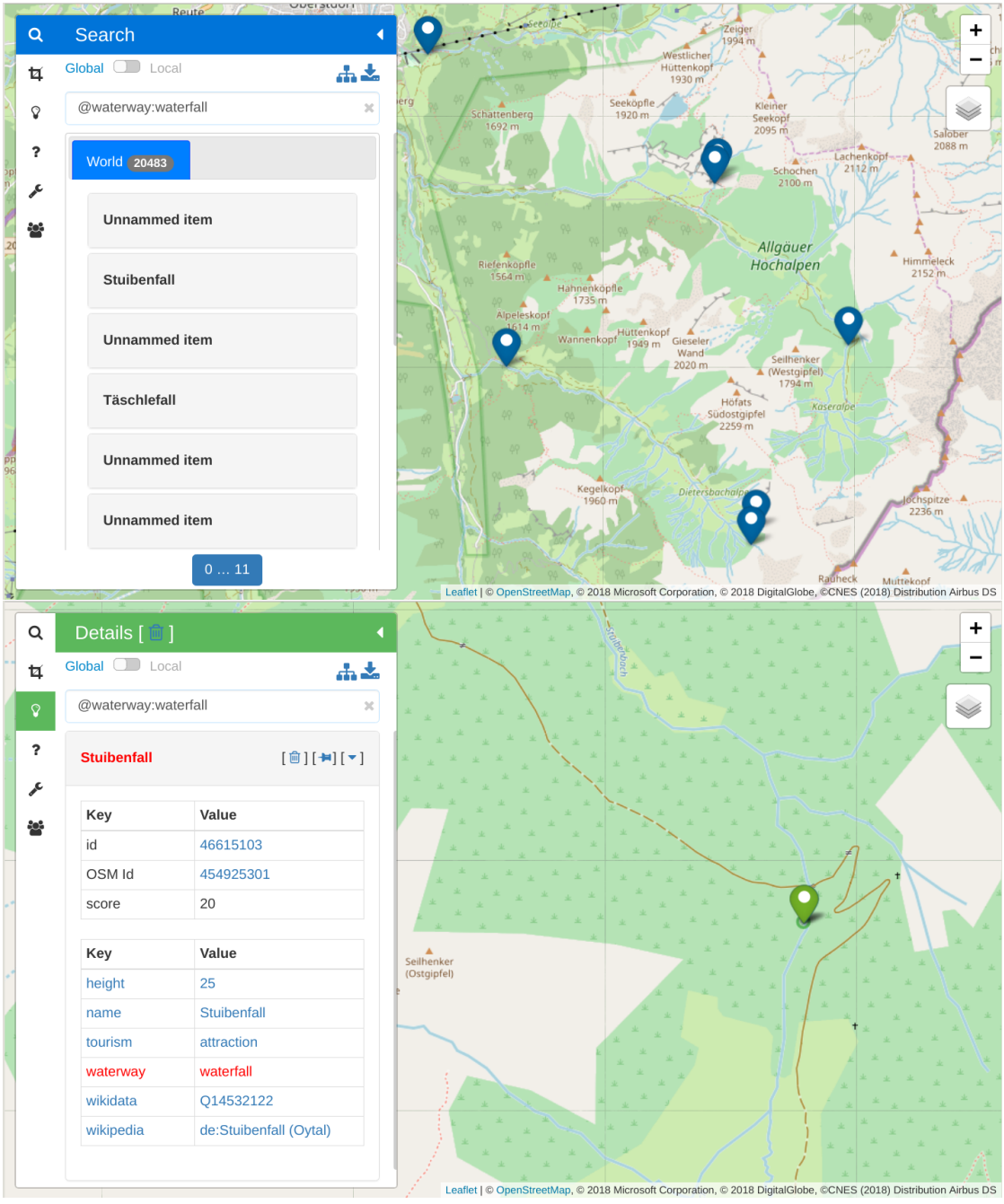
We then zoom in to Europe:



And further to Bavaria and the Allgäu near Oberstdorf.



On a low zoom level item markers are drawn and we click on the item representing the waterfall “Stuibenfall”.



4 Data Structures and their Implementation

In the following sections we will describe some of the data structures that are especially important to the workings of OSCAR. We start with a discussion of our serialization library where we present some important design choices. We then give a detailed introduction of the data structures used to represent cell-based queries and efficient implementations of the various cell operations described in former chapters. We then describe the architecture of OSCAR and all of its components needed to create and query our search data structure.

Notation The storage layout of a data structure often has a great impact on its serialization and usage speed and also influences the provided functionality. Consider for example a list of integers that we want to store on disk. Having random access to these integers greatly reduces our options on how to store them. If on the other hand linear access is needed then many compression schemes may be applied to the list. The exact data layout in memory and on disk is therefore an important information for an implementer. We give short layout descriptions for most data structures introduced in this chapter. We use a language similar to the C++ programming language which hopefully is self-explanatory. Table 4.1 gives an overview of the basic building blocks.

uint<n>	Unsigned integer with n bits
sint<n>	Signed integer with n bits
vuint<n>	Variable length integer storing up to n bits
vsint<n>	Variable length integer storing up to n bits
[...]	Array of Objects
struct NAME {...};	Structure named NAME grouping multiple objects consecutively
union NAME {...};	Structure named NAME interpreting the same data in different ways
TYPE NAME;	Variable of type TYPE named NAME
TYPE{VALUE}	Initialize type Type with value VALUE
VARIABLE = VALUE	Assign variable VARIABLE the value VALUE
_?	A place holder for a variable argument

Table 4.1: Basic building blocks for storage layout descriptions

4.1 The Serialization Library

The data that we want to process is a lot larger than the available system memory. We therefore need a way to efficiently store data out of memory on a hard drive. In order to achieve this goal we developed our serialization library called *sserialize*. We chose to develop our own library in order to be able to tailor it to our needs. We are especially interested in the ability to use different storage back ends, do copy-free decoding and encoding as well as use memory mapped files as our main storage system. Development of *sserialize* started in mid 2011 whereas other libraries supporting copy-free decoding and encoding were released afterwards. Cap'n proto [38] in 2013 and FlatBuffers [108] in 2014. At its core is the class `UByteArrayAdapter` which provides an interface to access different kind of storage back ends. It has numerous convenience functions to help a user store complex data structures on disk. One important aspect is how the system decides which parts of a file should be loaded into memory. The simplest way is to explicitly tell the library which parts need to be loaded. This usually means that users of the library would need to take care of memory management themselves which is quite tedious. Instead it is better to let the storage back end do the memory management. We have implemented multiple storage back ends with varied characteristics. Most importantly is the file access using memory mapping. In this back end we do not take care of the memory management ourself. Instead we let the operating system do that task. Files are then loaded in chunks of system pages into memory. The advantage is that all the available system memory can be used if possible while not interfering with other processes. The operating system can simply remove parts of the file from memory to free space if another process

requested more memory. Furthermore the operating system tracks the usage of the pages which should result in good page eviction decisions.

4.1.1 Posting Lists

At the heart of an information retrieval system one usually finds so-called posting lists as a part of other data structures like an inverted index. These are usually represented by lists of integers which may be stored in ascending order. The class `ItemIndex` is an interface for such a sorted list of unique unsigned integers. There are various compression schemes available which we will describe in short below. One important aspect of the implementation is that the compression scheme of the result of a set operation is the same as the scheme of the inputs. This can actually lead to a smaller memory footprint if the result compresses better than the input.

Approximation with Low-order Polynomial A rather simple approach to compress the data is to approximate the set using a low-order polynomial. One only has to store the difference to the polynomial for each entry. This can be further improved by using splines instead of polynomials. This essentially partitions the input data set into smaller sub sets and applies the polynomial approximation on these smaller sets. The downside of this approach is that compression rates may be rather small. On the other hand one does not loose the random access capabilities of the naive solution.

Run-length Encoded Bit Vector The Run-length Encoded Bit Vector encoding is an efficient storage scheme for bit vectors. It is based on the "Word Aligned Hybrid Index" described in [111] which is based on the observation that CPUs need the same amount of time to process data smaller or equal to the word size of the processor. As a first step the bit vector is cut into chunks of 31 bits. Then a simple run-length encoding scheme is applied on these chunks. Each word of the compressed data then stores whether it encodes a run of zeros or ones or if it encodes a chunk of the original data. Compared to [111] we store this marker bit in the least significant bit of a word. The advantage is that a variable length integer encoding can be applied on top of the bit encoding. See 4.1 for a description of the storage layout. To compute an intersection of two encoded lists one may simply decode each word on the fly or use the run-length information to speed up the computation.

Run-length Delta Encoded Integer Vector In the run-length delta encoding scheme we first compute the gap between successive entries of an integer list.

```
union Entry {
  struct TypeInfo {
    uint<1> type;
    uint<31> dummy
  } type;
  struct FullWord {
    uint<1> dummy; // == 0
    uint<31> data;
  } full_word;
  struct RleWord {
    uint<1> dummy; // == 1
    uint<1> type;
    uint<30> amount;
  } rle_word;
};

struct WAH {
  uint<32> size;
  [Entry] entries;
};
```

Code Listing 4.1: Storage layout of a word-aligned hybrid index. A single entry is 32 bits wide.

Then a run-length encoding scheme is applied on these gaps: The least significant bit acts as marker bit to indicate whether the next code word encodes multiple source entries or a single entry. Runs of gaps of the same value are encoded in two code words. First the length of the run, then the value of the gaps. See 4.2 for a description of the storage layout. A variable length integer encoding scheme may be applied on top as well. It's possible to speed-up the decoding phase using SIMD operations as described in [89]. Another faster alternative surpassing the patented encoding scheme VARINT-G8IU introduced in [103] is described in [69]. However the latter format needs to know the number of integers to encode beforehand since they store control information – the number of bytes a number needs – and the data in separate positions.

Elias-Fano Encoding The Elias-Fano representation introduced in [36] splits the entries l of a list L into $k = \left\lfloor \log_2 \frac{\max(L)}{|L|} \right\rfloor$ lower bits and the remaining $\{l_u = \log_2(l) - k\}$ upper bits. For the upper bits gaps are computed and the result is stored using unary coding. The lower bits are stored explicitly using k bits per entry. An entry can now be computed by adding up the gaps of the upper bits together with the entry of the lower bits array. The exact storage layout is shown in listing 4.3. [109] extended this encoding scheme with skip pointers in order to find the smallest entry i such that $x_i \geq b$ for a given bound b. This is

```
union Entry {
  struct TypeInfo {
    uint<1> type;
    uint<31> dummy;
  };
  struct Delta {
    uint<1> dummy; // == 0
    uint<31> value;
  };
  struct RunLength {
    uint<1> dummy; // == 1
    uint<31> length;
  };
  struct RunLengthDelta {
    uint<32> delta;
  };
};
```

Code Listing 4.2: Storage layout of a run-length delta encoded integer vector

especially helpful to do list intersection for large lists. The impact of this enhancement to OSCAR's list processing is likely very small since most posting lists store less than 1000 elements.

(Patched) Frame of Reference Encoding See section 2.4 for an introduction of (Patched) Frame of Reference encoding. In short we store the integers in blocks of fixed size, each block with a fixed number of bits per entry. Numbers that don't fit in the fixed amount of bits are stored somewhere else. We implemented a simple version with a focus on good compression ratio. Note that in our case the input data is a sorted list of unique integers and hence the difference between two consecutive elements is always larger than 0. We therefore only store the gaps of the list. We also do not store the minimum or maximum of a block since these can be computed from the previous block and the explicitly given bit length. Outliers are encoded using a variable byte length encoding. The presence of such an encoding is indicated by a 0 within the fixed bit width part. An input posting list is partitioned into equally sized blocks each having their own bit width. The compression ratio mainly depends on the chosen block size as well as the bit width within the blocks. We define 32 possible block sizes and compute in $\mathcal{O}(n)$ the optimal parameters for a given input list.

```
struct unary_int<n> {
  [uint<1>{0}] number; //n times a 0
  uint<1> stop = 1; //the end of the number
};
struct Index {
  [uint<n>] lower; //n=1..31
  [unary_int<_?>] upper;
};
```

Code Listing 4.3: Storage layout of a run-length delta encoded integer vector

```
struct Block {
  [uint<n>] fixed; // n=1..32
  vuint<32> outliers;
};
struct Index {
  uint<32> size;
  uint<5> blocksize;
  [uint<5>] blockbits;
  [Block] blocks;
};
```

Code Listing 4.4: Storage layout of the Patched Frame of Reference compression scheme

4.1.2 Reducing Storage Space

We can further decrease the total storage space by automatically selecting the best compressing index type for a given posting list resulting in an index with mixed types. Another option available is to use a storage back end with transparent compression support.

4.2 Cell Based Query Representation

OSCAR's main ingredient is the usage of a cell arrangement to efficiently compute a result for a given query. A fast representation of a subset of a cell arrangement is therefore a necessity. We use two different representations to answer a query. The class `CellQueryResult` represents a fully defined subset of a cell arrangement which means that each cell of a result is either full-match or partial-match and all partial match indexes are available. Its in-memory storage layout is described in listing 4.5. It is used together with an index containing all the necessary posting lists and the cell arrangement of which it is a subset. A set operation of two `CellQueryResult`s can easily be computed linear in the number of cells if no partial-matched cells need to be processed. Processing a set operation between two partial-matched cells is linear in the number of matched elements of the involved cells.

However this may incur a pointless overhead for complex queries. Consider for example the query "*(@building @highway) #Germany*" which computes all highways that are also buildings in Germany. A simple implementation would first execute the query "*(@building @highway)*" which has rather large input operands each having mostly partial-matched cells. Delaying these partial-matched cell operations until the end would greatly speed up the processing since it would then be possible to eliminate all cells that are not in Germany before processing the partial-matched cells.

The `DCQR` class implements such a delayed set operation scheme. In this scheme each cell has the full set operation history. In essence this means that each cell has a tree resembling the operations computed on a `CellQueryResult`. Its in-memory storage layout is described in listing 4.6. The trees are stored in a single array where each node is represented by a single 64 bit integer. The exact semantics of this integer are defined by the `FlatNode` class. The operation tree of a cell is stored consecutively in this array providing a cache-efficient access pattern.

Since the cells are stored with increasing ids a simple merge algorithm can be used to compute the set operation of two `DCQR`. In the following we will only give a description of the intersection operation. Other set operation types can be computed analogously. For the intersection operation it is enough to compute results for the cells that are in both operands. If both cells are full-match, then

we store this information in the cell description array. If one is full-match and the other is not, then we copy the information from the other operand. If both operands are defined by an operation tree a new operation tree needs to be created with a new intersection node as the root node. The trees of both operands are then copied as subtrees to the result. Thus a set operation can be computed with a single linear read of the cell descriptions and cell trees arrays. Finally all operation trees have to be applied in order to retrieve the result. We can compute this in parallel since all trees are independent of each other. See figure 4.1 for a graphical visualization.

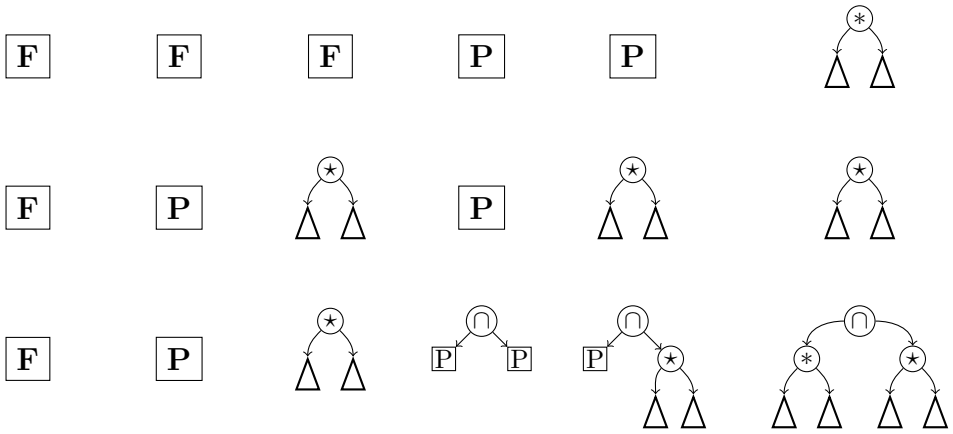


Figure 4.1: Intersection operation on a DCQR: We compute in a single linear sweep the relevant cells together with their operation trees. The top row shows the possible types of the first operand, the middle row shows the types of the second operand and the bottom row shows the result of the intersection between the two operands. An **F** indicates a full-match cell, a **P** indicates a partial-match cell, $*$ and $\star$ indicate any kind operation.

Partial-Match Posting Lists Posting lists of partial matches may refer directly to the items using their id or indirectly through the cell's item list. We refer to the first scheme as global (item) ids and the latter as (cell-) local (item) ids. In the case of local ids one has to map them back to global ids before retrieving an item. The benefit on the other hand are posting lists with smaller gaps, numbers and fewer differing posting lists resulting in less storage space and faster processing speeds for queries with large intermediate operands and rather small results. For the planet data set with substring search this results in posting list data size of 51.6 GiB for global ids versus 31.7 GiB for local ids. This is especially useful for

```
1  struct CellDescription {
2    uint<1> fullMatch;
3    uint<1> fetched;
4    uint<30> cellId;
5    union IndexDescription {
6      ItemIndex idx;
7      uint<32> idxId;
8    }
9  }
```

```
10 struct CellQueryResult {
11   [CellDescription] cells;
12 }
```

Code Listing 4.5: In-memory storage layout of the `CellQueryResult` data structure used to represent a subset of a cell arrangement

our web-based demonstrator which usually fetches only a small subset of all items in the result set and hence the processing time of the mapping step from local ids to global ids is negligible.

```
union FlatNode {
  uint<64> raw;
  struct Common {
    uint<4> type;
    uint<60> dummy;
  } common;
  struct OperatioNode {
    uint<4> type;
    uint<60> childB;
  } opNode;
  struct PartialMatchNode {
    uint<4> type;
    uint<28> cellId;
    uint<32> pmIdxId;
  } pmNode;
  struct FullMatchNode {
    uint<4> type;
    uint<28> cellId;
    uint<32> padding;
  } fmNode;
  struct FetchedIndexNode {
    uint<4> type;
    uint<60> internalIdxId;
  } fetchedNode;
};
```

```
struct CellDescription {
  uint<1> fullMatch;
  uint<1> hasFetchedNode;
  uint<28> cellId;
  uint<32> pmIdxId;
  uint<2> dummy;
  uint<32> treeBegin;
  uint<32> treeEnd;
};

struct DCQR {
  [CellDescription] cells;
  [FlatNode] trees;
  [ItemIndex] idx;
};
```

Code Listing 4.6: In-memory storage layout of the `DCQR` data structure used to represent a subset of a cell arrangement

4.3 From Query to Cells

The input to OSCAR is usually a string describing a query in a more or less human readable form. See appendix 8.2 for the full definition of OSCAR's query language. This is parsed into an abstract syntax tree without any further modifications like stemming, normalization, or even complex natural language processing facilities. The advantage is that "you get what you request" since it is obvious for the user what the system computes based on a given query string. The downside is that the query language is difficult to understand for non-technical users. To compute the cells of the query we simply execute the processing statements of the as described in figure 4.2. The following sections will describe the individual processing statements in detail.

4.3.1 Textual Queries

Textual queries can be answered using a data structure that maps strings to arbitrary data. We use a simple generalized suffix tree-like scheme as describe in section 2.2 that stores for each entry all matching cells and items. Using a compressing text index would only slightly reduce the storage space since textual data itself only accounts for less than 5% of the total search structure storage space including the posting list. To answer a query we simply do a search in this data structure and return the set of cells matching the item. In the case of a region query we return a list of full-match cells, a item query returns a list of partial-match cells with points to item lists and a query for both return the list of full-match cells and the list of partial-match cells with the pointers to the appropriate item lists. This is exactly the data needed to initialize a `CellQueryResult` object.

4.3.2 Spatial Queries

Compared to textual queries computing the result of a spatial query is more involved. We use three different data structures to represent the spatial data. At the lowest level we use a Delaunay Triangulation to represent the cell arrangement. This triangulation is used for path and point location queries. The cell graph is one layer above and represents cells of the cell arrangement. It is mainly used for the cell dilation query. Another level above resides an explicit representation of the Inclusion-DAG which is used to answer all polygon queries by interpreting it as an R-tree. See figure 4.3 for a graphical visualization.

Precision Issues

Our on-disk representation uses fixed precision numbers to store all spatial data. This however introduces a problem when the input data produces spatial data

1. Parse query into Abstract Syntax Tree
2. Execute leaf node operations:

   Text query Use cell based suffix array

   Polygon Use the Inclusion-DAG

   Point, Path Use the Cell Arrangement and Cell Graph
3. Execute inner node operations:
   - Set operation:

     CQR Compute result immediately

     DCQR Delay computation, add operation to cell operation trees
   - Complex spatial operation:
     a) Execute delayed operations
     b) Execute complex spatial operation:

        Between, Cardinal Direction Construct polygon for polygon intersection test

        Nearby Use the Cell Graph
4. Finalize result:
   a) Execute delayed set operations
   b) Remove empty cells
   c) Map local item ids to global item ids
5. Compute subgraph
6. Compute union of cells to retrieve items within selected cells

Figure 4.2: Full execution pipeline of OSCAR from input query to output subgraph.

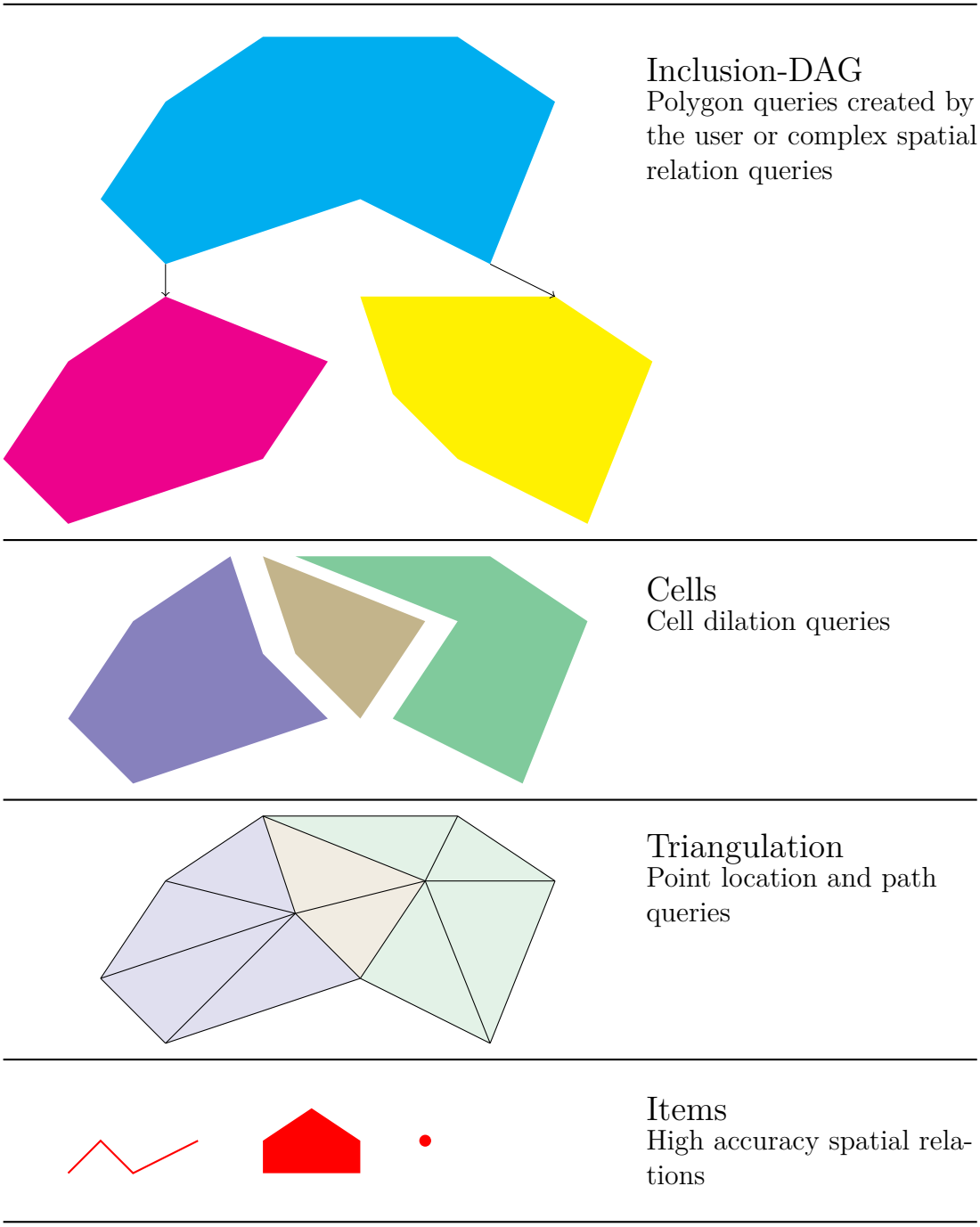


Figure 4.3: Data structures used for the various spatial relation queries.

with higher precision. One example are intersection points of polygonal line segments. The endpoints of the segments may very well be representable with our defined precision - the intersection however is not. To keep the fixed precision number, we have to snap the intersection points in a way that they are representable and do not alter the geometry too much. If we don't snap the points, but simply store points with the lower precision then the resulting static triangulation has a geometry that does not match the topology. Even worse it very likely is not a triangulation anymore. We can remedy this by changing the algorithms working with the triangulation. This is especially the case for the triangulation walking algorithm. One possible solution is to switch to the topology if the algorithm behaves strange to get out of a region where the geometry is incorrect. This however slows down the straight line walking algorithm and is cumbersome to implement. Additionally we may visit triangles of cells that do not intersect the path or miss intersected cells. As an alternative – which is the default in OSCAR – we try to snap the points in-memory while trying to keep the topology the same. CGAL already provides an implementation of the (iterated) snap rounding algorithm described in [56, 48]. The absolute resolution of our static representation is below 1 cm and thus inaccuracies introduced by snapping the points are not relevant for our data set considering that the input data has an accuracy of around 3 m and constraint edges are relatively short. Hence in practice the simpler algorithm shown in code listing 4.7 is good enough. Note that input points are already snapped by definition which is why only intersection points of constrained edges need to be snapped. For the planet data set this algorithm runs in 20 seconds.

Projection Issues

Our input data is stored as two floating point numbers representing coordinates in the WGS84 coordinate system. Our current implementation directly builds a 2-dimensional triangulation based on these coordinates – in essence a equirectangular projection. In theory this may introduce problems due to incorrect intersection points of polygonal line segments. Fortunately most regions of our input data is densely sampled and hence the introduced error is negligible. Furthermore the resulting triangulation is not a Delaunay triangulation on the sphere which is a property that we do not necessarily need. In the future we will integrate our library from chapter 6 to produce accurate triangulations on the sphere. Additionally this would also solve the precision problems discussed in 4.3.2.

4.3.3 Quality of Results

The quality of a result depends on the directives used in the query. The text search part of OSCAR returns an exact result containing only matching items.

```
1  Triangulation tr; //the triangulation
2  Queue q; //Edges, sorted by length
3  Edge e, xe; //Constrained edges
4  while(q.size() && q.top().length > threshold) {
5    e = q.pop();
6    xe = firstIntersectedConstraint(tr, e);
7    if xe {
8      p = snap(intersectionPoint(xe, e))
9      tr.remove(xe)
10     q.insert({(p, e.begin), (p, e.end),
11               (p, xe.begin), (p, xe.end) })
12   }
13   else {
14     tr.insert(e)
15   }
16 }
```

Code Listing 4.7: "Snapping" algorithm used to create a fixed-precision triangulation which can be stored on-disk.

Spatial queries on the other hand return approximate results. Queries like the betweenness and cardinal direction query construct a polygon to answer the request. The size of this polygon directly influences the quality of the approximation. Very small polygons return an exact result regarding polygon-item intersection. For very large polygons only the bounding-box of the polygon and the bounding-box of the cell is taken into account. The thresholds for the accuracy can of course be changed at any time.

The approximation quality of queries using the cell arrangement like the cell dilation or path corridor operator depend on the size of cells and the type of the cell arrangement. If cells in the vicinity of a path or dilated cells are rather small then the result is quite accurate. This however may increase the processing time of a query since more cells are involved.

4.3.4 From Cells to Items

For most usage scenarios it is enough to provide the matching items partitioned by the cells together with the subgraph of the Inclusion-DAG. However we may also be interested in all items of a result. To achieve this we have to unite the posting lists of each cell. Our implementation currently does not take the size of the cell indexes into account which may produce a bad merging order if for

example we constantly merge a large index with a small one. [82] proposed an algorithm to efficiently and practically compute such a good merging order. They compared their implementation with other state-of-the art sorting algorithms and found that a nearly-optimal merging order resulted in a speed-up of 20% to 100%. This is of course only useful for a sparse result. For results that are made up of a large fraction of all items we use a bit vector to represent it. To this end each implemented index compression scheme supports a specially crafted decoding function to put its content into a special bit vector class. We can then recreate a compressed index from this bit vector – again using optimized functions depending on the compression scheme.

4.4 OpenStreetMap Data to OSCAR

In order to query an OpenStreetMap data set we first have to convert it to our own format. As a first step we transform the input data into our own storage formats and data structures. We then compute various search structures based on this representation. The following sections cover the whole preprocessing pipeline with an overview shown in figure 4.4.

Data Base Creation We first extract all relevant items and regions and store this data in our own data format on disk. To this end we employ a library [3] to read the binary format of OpenStreetMap data dumps. Reading the whole data of a planet dump into main memory is prohibitive which is why we process the data in multiple rounds. Furthermore we extensively use offset arrays to keep memory fragmentation on a low level. We start by collecting all relevant regions and construct a quad-tree like data structure to support fast point-in-polygon queries. In contrast to a quad-tree we can use different branching factors per node with the root node ranging between 100^2 and 3000^2. Subsequent levels are usually split into 2^2 children. We refine a tree node if its diameter is smaller than a given threshold (default of 10 km) and if it contains or intersects any regions. Each internal node stores a set of regions that fully cover the nodes' associated bounding box whereas leaf nodes store intersecting regions. The tree itself is stored in an offset like data structure. For the planet data set this boils down to about 215 MiB memory constructed in 472 seconds. We use this tree for some data cleaning purposes but also to assign a triangle of the cell arrangement its covering regions and hence its cell. Among many things the data cleaning involves removing regions that do not contain relevant items for which we have to find for each relevant item all covering regions. We construct the final cell arrangement using the constrained Delaunay triangulation data structures with a lazy exact predicates and exact constructions kernel from CGAL [91]. Note that the use of the exact constructions kernel results

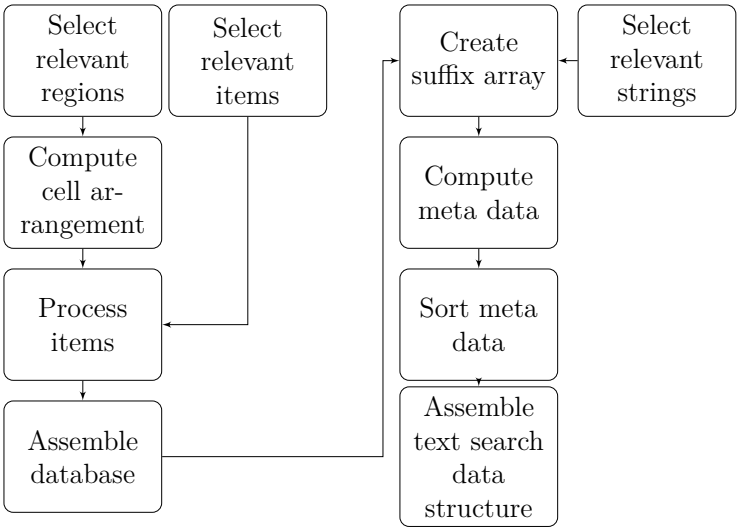


Figure 4.4: Stages of preprocessing and query processing.

in a higher processing time and larger memory usage. However our application consistently crashed using the in-exact constructions kernel while processing the planet data set. See section 7 for a possible solution. Depending on the selected options we then refine the resulting triangulation and the cell arrangement. Finally we process the input data in multiple rounds to extract all relevant items and their relation to the regions and their place within the cell arrangement.

Cell-based Text Index The second phase builds user specified indexes based on the data base of the first phase. OSCAR supports cell-based text query data structures, grids, R-trees and simple inverted indexes. All indexes are constructed in-memory with the exception of the cell-based text index which can also be created out-of-memory:

We first compute a generalized suffix tree stored in an array containing all relevant strings. In the second phase the mapping of strings to items and cells is stored in a file on disk. This file is sorted in the third phase. The last phase computes in a single linear sweep a compressed version using our own static out-of-memory data structures.

4.5 User Interfaces

Apart from our web-based demonstrator we also implemented a command-line client and a Qt-based desktop application. All applications use the C++ library

liboscar to access OSCAR's data structures. We also implemented a small wrapper library for Java applications which is used in the comparison with Lucene and MG4J. Additionally a Python interface is available with which one can interactively work with some of OSCAR's data structures. In the following we would like to give a short description of the architecture of our web-based demonstrator which is what most users interact with.

Web-based Demonstrator Our web-based demonstrator consists of a client running in a browser and a back end running on a server. The back end uses the C++ framework CppCMS [23] to provide a stateless API to the client. The client is implemented as a single page application which dynamically loads all content using JavaScript. Data is usually transferred using JSON [25] as encoding for structures. We also experimented with binary transfer of search results. In particular the indexes of a cell can be transferred without decoding on the back end server but instead on the client. This is of advantage if a user wants to explore the whole data set since then only the cell indexes are fetched on demand whereas the subgraph of the Inclusion-DAG is available at once. This however transfers more data than necessary if only small parts of the result are of interest to the user. Thus our public facing demonstrator does not use the binary transfer mode but instead dynamically loads and caches the parts requested by the user. The downside is more load on the server since we do not cache search results yet and latency on the client side while browsing through the data. One important aspect of this mode of operation is that it emphasizes on fast cell computation instead of retrieving all items of a result. Items are fetched for a rather small subset of the result based on the interaction with the user.

5 Experimental Evaluation

We conducted extensive benchmarks to underline the performance of OSCAR. Our benchmark setup is introduced in section 5.1 which hopefully gives the reader the means to replicate our results.

We start with a discussion of some performance aspects of our serialization library in section 5.2 where we show the impact of the storage abstraction layer. With the basics set we turn our focus to OSCAR beginning in section 5.3. We first analyze the performance of the preprocessing step to produce the data sets used for querying. The following sections discuss the query performance with respect to the used cell refinement (5.6.2), delayed cell computation (5.6.3), query type (5.6.1), number of processing threads (5.6.3) and index compression scheme (5.5). We also conducted some benchmarks on a mobile phone to show OSCAR's ability to work in resource restricted environments. Furthermore section 5.8 shows the impact on storage space and query performance for more regular cell arrangements. We conclude with a comparison of OSCAR to Lucene and MG4J in section 5.9 where we show the competitiveness of our approach compared to these two fast text indexes.

5.1 Setup

In the following we describe our experimental setup in case the reader is interested in reproducing our results.

5.1.1 Hardware

Preprocessing and server query performance were measured on a powerful server. We also used a mobile device to show OSCAR's ability to cope with resource constrained environments. The detailed hardware specifications of our test environments are listed in table 5.1.

5.1.2 Software

Our benchmarks are based on 4 different projects bundled in a single umbrella project which can be obtained from [12]. The umbrella project is organized as follows: the folder *progs* contains all programs needed to run the benchmarks and the analysis whereas the folder *data* is a git submodule containing the data

	Mobilephone	**Server**
Model	Sony Xperia V	Transtec custom built
CPU	Qualcomm MSM8960 Snapdragon S4 Plus	2 x Intel Xeon E5-2650v4
Cores physical/logical	$2/2 \times 1.5$ GHz	$2 \times 12/24 \times 2.2$ GHz
RAM	1 GiB	768 GiB
SSD	Class 10 SD card single	2×960 GB Samsung MZ7KM960HAHP in raid0
HDD		3×4 TB HGST HDN724040ALE640 in raid0
Operating System	Android 4.4	Ubuntu 17.10

Table 5.1: Specifications of our test environments

generated by OSCAR and the subsequent data analysis. Within the *data* folder is a submodule named *private* which contains the used queries. Note that we cannot publish the queries since these may contain sensitive information which a user may have accidentally entered in our search engine. Interested readers may contact us in order to obtain a copy of this data. However we publish [11] the resulting benchmark data which is used in our analysis.

Set Up The folder *progs* contains a script `setup.sh` to build all necessary libraries and programs. By default it compiles OSCAR with 3 different compiler options. The `debug` build has no optimizations enabled at all and is only used for debugging purposes. The `lto` and `ultra` builds on the other hand use processor specific compiler optimizations together with link time optimization. The link time optimization is especially useful since *sserialize* and OSCAR make extensive use of the private implementation pattern. During the link time optimization phase the compiler has access to the whole program code and is able to eliminate many indirections caused by this pattern. The difference between the `lto` build and the `ultra` build are the build parameters passed to OSCAR and *sserialize*. The `ultra` build dynamically disables reference counters of some heavily used data structures to reduce the synchronization overhead in environments with many threads. Additionally the support for non-contiguous storage back ends is disabled to reduce its abstraction overhead.

Switch	Argument Values	Description
-t	store, oscar, lucene, mg4j, queries, indexes, oscar-bench, index-bench, …	Selects what should be created
-s	Source name	Base name of input data set i.e. planet for planet.osm.pbf
-c	disconnected,nocellsplit,cellsplit1k,celldiag5k	Name of the cell refinement configuration
-b	debug, lto, ultra	Name of the build configuration
-ot	prefix, substring, substring-celllocalids	Selects the type of search that OSCAR supports and if posting lists refer to cell local ids (cf. 4.2)

Table 5.2: Program options of the creation script

Benchmarks All our benchmarks are available through the single script `create.sh`. See table 5.2 for a description of the command line parameters. The script `thesis-create.sh` executes all relevant benchmarks. It first creates the stores, search structures and text indexes. These are then used to execute the benchmarks. In order to replicate all results one needs at least 512 GiB of system memory and 30 TiB of storage space.

Analysis Most of the tables shown in this chapter are produced by the script `analyze.sh`. Additionally it also post-processes the raw data of our benchmark runs such that it is possible to produce nice looking graphs solely using TEX's tikz package.

5.1.3 Measurement Methodology

We used the GNU `time` program [46] to measure the total execution time, peak memory usage and CPU utilization. CPU utilization may be larger than one which indicates that more than one CPU was used during the computation. Peak memory usage does not reflect the used memory in reality but rather the memory requested from the system. The standard memory allocator uses the *libc* allocator which manages its own memory pool. If a program releases memory back to the allocator then the memory region is put back into the memory pool. It is possible to force the allocator to release freed memory back to the system using `malloc_trim`. We did not use this since it is not clear when to call this function and

calling it too often would likely degrade the performance. An unfortunate result are slightly different memory usages between multiple runs.

5.2 Storage Library

In the following we compare the overhead of using our storage library *sserialize* with simple C structures or even just integers. We furthermore explore the impact of the storage back end abstraction and compare our integer packing implementation with another state-of-the art implementation.

5.2.1 Abstraction Overhead

sserialize uses a simple encoding scheme to store integers and strings on disk. Fixed-length integers are stored in little-endian order which is the endianness of most modern CPUs. Variable length integers are stored using 7 bits per byte for the data and 1 bit to indicate if another byte follows. There are of course other more complex data structures to support integers of variable bit sizes, Huffman-Codes or transparent compression. Support for multiple storage back ends is implemented using C++ virtual functions resulting in a virtual function dispatch for each data access.

We compare the performance of our implementation with a simple array and a `std::vector` using bounds checking. To that end we first store 10^9 numbers and then read them back again. We repeat this 100 times and take the mean of all runs. We refer the reader to table 5.3 which we will discuss in the following. The random column shows the timings for numbers chosen uniformly at random in the range of the given data type. The sequence column shows the timings for the sequence $s_n := s_{n-1} + 1$. We also compare two different build configurations. Both builds are compiled with link time optimization enabled. The `lto` build supports all back ends available in *sserialize*. The `ultra` build only supports back ends with contiguous storage. This limitation makes it possible to eliminate the virtual function dispatch needed when fetching data.

If we compare the machine code generated by the compiler then we will find the following code executing for each entry: The unchecked array compiles to a single `mov` instruction taking about a cycle [43]. The `pack vuint` is inlined in the loop. The `UBA uint` variant compiles to an inlined bounds check and a single `mov` operation. The other operations compile to function calls.

Comparing rows `UBA uint` and `pack uint` in column `ultra` we can see that the bounds checking reduces the throughput by about 40%. The virtual function dispatch is barely visible as can be seen when comparing column `lto` to column `ultra` in row `UBA uint` and `UBA vuint`. Variable byte length encoding on the other hand reduces the throughput by a factor of 8 to 10.

```
1 std::vector<uint32_t> input;
2 uint32_t sum = 0;
3 for(std::size_t i(0); i < testLength; ++i) {
4   put(input[i]);
5 }
6 for(std::size_t i(0); i < testLength; ++i) {
7   sum += get(i);
8 }
```

Code Listing 5.1: Code to benchmark encoding and abstraction overhead. `put()` and `get()` are storage specific functions.

		random		sequence	
Container	Bits	`lto` $\left[\frac{M}{s}\right]$	`ultra` $\left[\frac{M}{s}\right]$	`lto` $\left[\frac{M}{s}\right]$	`ultra` $\left[\frac{M}{s}\right]$
`pack uint`	32	879.1	850.1	848.1	843.5
`pack vuint`	32	152.3	153.7	193.1	191.8
`UBA uint`	32	501.2	508.3	507.7	505.4
`UBA vuint`	32	103.7	105.2	121.9	121.7
`std::array`	32	872.3	876.3	878.7	880.9
`std::vector`	32	332.0	320.6	321.0	319.9
`pack uint`	64	440.2	433.1	432.9	432.9
`pack vuint`	64	59.3	60.3	177.2	177.2
`UBA uint`	64	393.3	394.2	395.7	395.1
`UBA vuint`	64	44.8	45.6	112.6	112.6
`std::array`	64	437.6	442.2	441.5	441.5
`std::vector`	64	204.6	199.6	199.1	199.3

Table 5.3: Encoding and abstraction overhead of our storage abstraction compared to a plain `std::array` with unchecked access and a `std::vector` with checked access. See paragraph 5.2.1 for an in-depth discussion.

5.2.2 Integer Packing

Integer packing and unpacking is heavily used in the FoR and PFoR index compression schemes where we need to code a set of integers with b bits into an array of bytes. The performance of this packing is heavily influenced by the implementation, compiler and processor. Comparison of different implementations is difficult without access to the code base. Fortunately some researches publish their code such that it is possible to do some comparisons. Specifically we compare the unaligned integer packing implementation from [68] with our variant (see table 5.4). Note that we store integers in big endian while our test platform is a little-endian system which is why we need to do a byte swap to retrieve our integer. Our implementation is based on the idea that loading a machine word has the same cost as loading a single byte. Additionally we can do the byte swapping using a single `movbe` instruction. The code from [68] on the other hand extracts all relevant bits from each byte using shifts and masks. The data is stored in blocks of 16 integers with a special function for each bit width. The advantage of this is an improved instruction parallelism due to static data dependencies. Our implementation has a static data dependency within a block as well but the throughput of the `movbe` instruction is lower than a simple `mov` or `load` [43, p. 216]. As a result our packing code is slower than theirs by a factor of 1.2 to 1.7. The unpacking function on the other hand is either faster or only slightly slower.

Bits	pack $\left[\frac{M}{s}\right]$		unpack $\left[\frac{M}{s}\right]$	
	sserialize	fastpfor	sserialize	fastpfor
1	2071	2427	2292	1637
2	2036	2367	2238	2244
3	1679	2340	2155	2120
4	2008	2384	2197	2181
5	1578	2284	2108	2145
6	1500	2222	2083	2108
7	1535	2182	2040	2074
8	2005	2287	2089	2062
9	1200	1989	2009	1953
10	1235	1998	1988	1993
11	1190	1922	1953	1878
12	1215	2049	1884	1949
13	1168	1885	1839	1776
14	1141	1866	1888	1885
15	1150	1796	1857	1773
16	1637	2196	1852	1847
17	1108	1732	1738	1776
18	1084	1723	1730	1759
19	1072	1650	1701	1676
20	1080	1625	1661	1736
21	1059	1571	1599	1633
22	1031	1553	1665	1623
23	1048	1469	1639	1547
24	1091	1541	1649	1651
25	1003	1402	1609	1527
26	1023	1421	1598	1568
27	984.3	1363	1583	1478
28	999.9	1358	1497	1507
29	966.9	1294	1549	1448
30	974.1	1275	1538	1473
31	941.4	1239	1510	1384
32	1202	1527	1528	1543

Table 5.4: Packing and unpacking integers of various bit widths into an array of bytes. We compare *sserialize* against the `fastunalignedpackwithoutmask_16` function from [68]

5.3 Source Data Sets and their Transformation

In order to query a OpenStreetMap data set we first have to transform it into the storage format of OSCAR. This transformation is split into two phases: data base creation and search structure creation. We use different extracts of the planet data set as source data. Specifically these are Baden-Württemberg, Germany, Europe and the whole Planet which we downloaded from [5] on the 18th of February 2018. We created 4 data bases for each data set with different options regarding the cell refinement step: no refinement, one with connected cells, one with cells with up to 1000 triangles and one with cells smaller than 5000 m in diameter. For the query benchmark we created two search data structures with global ids and local ids. The data structure obtained without any refinement and global ids essentially reflects the one introduced in [14]. There are of course some additional data structures available like the triangulation of the cells or the cell graph. If we enable the triangle refinement step with global ids then we get the data structures mentioned in [15]. The following sections give a more detailed description of the two preprocessing steps.

5.3.1 Data Base

The first phase produces the data base on which all search data structures are based upon. Various settings influence later phases like the selection of regions or the type of the cell arrangement. The statistics for our 4 used data sets are depicted in table 5.5. In general we can say, that the size of the database for the planet data set is around 70 GiB apart from the variant with small cells. The later introduces many new triangles and cells which increases the spatial data structures quite a bit. Processing time is always below 17 hours except for the small cells variant which has a gigantic processing time of almost 120 hours. This is solely caused by the slow and unoptimized triangle refinement and cell refinement steps. The effect can also be seen in the CPU utilization row where we normally have a utilization of 2.9 compared to more than 15 for all other refinement variants. Due to the larger triangulation data structure the memory usage also increases by more than 20 GiB of main memory. The number of cells also goes up by a factor of about 32 compared to a factor of at most 2 for all other refinement variants.

5.3.2 Search Data Structure

OSCAR has a multitude of options to configure the search data structures as well as the usage of computing resources. We used the settings listed in table 5.6. Case-insensitivity is achieved by converting all input strings to lower-case before indexing. This slightly reduces storage space since some entries of items are

	Ba-Wü	Germany	Europe	Planet
Source size [GiB]	0.4	4.2	20.6	39.7
Items [10^6]	8.0	55.9	324.4	593.6
Regions [10^3]	17.7	148.3	898.0	1,486.1
Keys [10^3]	7.8	20.5	47.3	69.2
Values [10^3]	675.0	4,121.3	41,547.6	77,390.7
	Disconnected Cell Arrangement			
Cells [10^3]	21.1	177.6	1031.7	1684.4
Time [h:m]	0:06	0:53	6:45	12:50
Memory [GiB]	6.0	20.6	121.2	243.3
CPU utilization [1]	11.0	14.6	31.8	13.7
DB Size [GiB]	0.7	5.4	36.0	69.5
Index Size [MiB]	9	71	401	691
	Connected Cell Arrangement			
Cells [10^3]	21.1	177.6	1031.9	1684.4
Time [h:m]	0:06	0:57	7:38	15:16
Memory [GiB]	6.0	20.5	116.3	231.7
CPU utilization [1]	12.9	15.7	16.3	15.4
DB Size [GiB]	0.7	5.4	36.0	69.5
Index Size [MiB]	9	73	395	691
	Maximal 1000 triangles per cell			
Cells [10^3]	34.0	288.7	1680.3	3134.3
Time [h:m]	0:06	1:04	7:56	16:03
Memory [GiB]	6.0	20.4	119.0	238.3
CPU utilization [1]	12.5	17.4	16.2	15.4
DB Size [GiB]	0.7	5.4	36.2	69.7
Index Size [MiB]	10	75	428	781
	Cell diagonal < 5000 meter			
Cells [10^3]	62.3	564.7	11362.6	50239.9
Time [h:m]	0:09	1:55	20:57	119:07
Memory [GiB]	6.0	21.2	125.1	266.3
CPU utilization [1]	5.2	4.3	3.9	2.9
DB Size [GiB]	0.7	5.5	38.4	80.7
Index Size [MiB]	11	88	659	1511

Table 5.5: Statistics of the four data sets used in our benchmarks. The index uses the Run-length Delta index format. Time and maximum memory usage were measured using the `time` utility.

Case-sensitive	no
Diacritic sensitive	no
Substring search	important values
Prefix search	all
Out-of-memory buffer	64 GiB
Threads	48

Table 5.6: Important settings set during the computation of the search data structures.

the same after this mapping. We furthermore add all strings with their diacritics removed. Substring search is enabled on the set of the important tags, all other tags only have prefix search enabled. Note that substring search entails exact, prefix, suffix and substring match whereas prefix match only entails exact and prefix match. The size of the resulting data structures are shown in table 5.8. The creation phase of OSCAR is mostly bound by the physical storage back end – in our case 3 cheap 4 TB hard disk drives. See table 5.7 for the resource usage of the search creation for various data sets and cell refinements. In order to process the planet data set in a reasonable amount of time one needs about 160 GiB of memory on our 48 threads machine. Note however the low CPU utilization of only 4 meaning that in average the equivalent of 4 CPU cores were fully used. However this does not mean, that using only 4 threads would achieve the same processing time since some steps in our preprocessing pipeline utilize all 48 threads. In order to further reduce the processing time one has to speed-up the storage back end for example by using a bunch NVMe SSDs. Our 3 drive back end can read and write data at a rate of about $350\frac{\text{MiB}}{\text{s}}$. Compare this to $2700\frac{\text{MiB}}{\text{s}}$ of a single Samsung SSD 970 Pro which is almost 8 times faster. In theory a 4 drive SSD raid0 configuration could achieve close to $10\frac{\text{GiB}}{\text{s}}$ which is more than 30 times faster than our storage back end. If we use the SSDs of our server we can already achieve a construction time of 1 hour 25 minutes for the Germany data set while reducing the memory usage from 70 GiB down to 18 GiB and the number of threads from 48 to 8.

	Ids	Ba-Wü	Germany	Europe	Planet
		Disconnected Cell Arrangement			
Time [h:m]	Global	0:11	1:08	8:39	16:03
	Local	0:11	1:03	8:26	14:36
Memory [GiB]	Global	36.5	69.8	99.6	154.1
	Local	36.6	70.6	104.2	155.9
CPU utilization [1]	Global	4.3	5.1	4.1	4.0
	Local	4.4	5.3	4.0	4.3
		Connected Cell Arrangement			
Time [h:m]	Global	0:11	1:05	9:11	15:28
	Local	0:11	1:03	7:51	14:22
Memory [GiB]	Global	37.4	71.6	101.6	161.3
	Local	37.5	72.4	106.7	156.1
CPU utilization [1]	Global	4.4	5.1	3.9	4.0
	Local	3.6	5.3	4.3	4.3
		Maximal 1000 triangles per cell			
Time [h:m]	Global	0:12	1:22	10:42	21:15
	Local	0:11	1:10	8:54	18:16
Memory [GiB]	Global	37.8	71.5	101.7	157.6
	Local	37.9	72.4	106.8	160.1
CPU utilization [1]	Global	4.3	4.4	3.4	3.3
	Local	4.5	5.0	4.0	3.6
		Cell diagonal < 5000 m			
Time [h:m]	Global	0:12	1:43	24:25	–
	Local	0:11	1:30	23:17	–
Memory [GiB]	Global	37.7	70.8	108.0	–
	Local	37.9	71.6	106.6	–
CPU utilization [1]	Global	4.0	3.7	2.1	–
	Local	4.2	4.4	2.2	–

Table 5.7: Preprocessing resources needed to compute a text search structure capable of substring search of the important values and prefix search on all key-value-pairs. The Id-column denotes the type of item ids used, see section 4.2. The index uses the Run-length Delta index format. We could not compute the text search structures for the planet data set using the maximum cell diagonal refinement since our system did not have enough storage space for the out of memory data structures.

	Ids	Ba-Wü	Germany	Europe	Planet
		Disconnected Cell Arrangement			
Text Search [GiB]	Global	0.3	2.6	17.8	33.0
	Local	0.3	2.4	16.0	32.2
Index [GiB]	Global	0.5	4.0	23.4	43.4
	Local	0.3	2.1	12.9	24.3
		Connected Cell Arrangement			
Text Search [GiB]	Global	0.3	2.4	15.7	33.1
	Local	0.3	2.4	15.7	32.4
Index [GiB]	Global	0.5	3.8	23.3	43.1
	Local	0.3	2.4	12.8	24.2
		Maximal 1000 triangles per cell			
Text Search [GiB]	Global	0.4	2.8	18.9	35.9
	Local	0.4	2.7	17.7	33.3
Index [GiB]	Global	0.4	4.1	26.4	51.6
	Local	0.3	2.3	15.6	31.7
		Cell diagonal < 5000 m			
Text Search [GiB]	Global	0.4	3.3	27.2	–
	Local	0.4	3.2	23.4	–
Index [GiB]	Global	0.5	4.4	34.0	–
	Local	0.3	2.4	22.2	–

Table 5.8: Statistics of our data sets for a text search structure capable of substring search of the important values and prefix search on all key-value-pairs. The Id-column denotes the type of item ids used, see section 4.2. The table also shows the effect of the refined cell arrangement. The index uses the Run-length Delta index format. We could not compute the text search structures for the planet data set using the maximum cell diagonal refinement since our system did not have enough storage space for the out of memory data structures.

5.4 Query Data Set

Typical information retrieval data sets like the TREC [86] data sets consist of documents, queries and respective answers. Unfortunately in the case of OpenStreetMap data this is not the case. The projects main search engine likely does not even store the queries due to user privacy concerns. We therefore rely on our own data set for which we stored all requests to our website demonstrator. We remove queries that were likely generated by scripts, had an empty result or were duplicates. We classified the queries into text-only and spatial queries and whether they contain set operations or not. Table 5.1 shows the resulting number of queries for each data set and query type. Spatial queries are a recent feature and their syntax is not intuitive resulting in a lower usage count. Additionally they are not needed that often since OSCAR's visualization allows for an easy inspection of large results.

Total	Ba-Wü	Germany	Europe	Planet
		Text queries		
Total	3670	4897	5734	6462
Exact	299	375	435	541
Prefix	40	72	87	118
Suffix	4	8	14	16
Substring	3633	4850	5687	6411
		Spatial queries		
Total	206	249	280	321
Path	10	10	10	10
Polygon	13	17	18	39
Rectangle	185	223	253	273
		Complex spatial queries		
Total	54	80	97	105
Nearby	14	20	22	24
Cell Dilation	24	28	31	28
Betweenness	0	7	6	6
Cardinal direction	18	26	40	49
		Set operations		
Intersection	2310	3304	3934	4583
Union	154	202	235	284
Difference	76	96	96	93

Figure 5.1: Characteristics of our input data set taken from the logs of our web-based demonstrator. Each entry gives the number of queries containing at least one of the query statements of the given row. We only use queries with a non-empty result.

5.5 Index Compression Benchmarks

OSCAR supports various index compression schemes described in section 4.1.1. Additionally posting lists of cells may refer to absolute item ids or cell local item ids. We use the planet data set with up to 1000 triangles per cell to compare these two options. In order to only benchmark the data structures and algorithms all data has been loaded into memory. Note that keeping the index in memory is always preferable over having to load the data from disk. If memory is tight one would therefore always choose the index with the lowest amount of storage space regardless of its processing speed.

We would like to refer the reader to table 5.9 which depicts the size of the planet data set index for the various index compression schemes. Interestingly the simple Patched Frame of Reference scheme has the best compression ratio. However decoding and encoding has a high overhead and hence the bad timings in the items column. Using a cell local index drastically reduces the storage amount while increasing query processing times only a bit. Which index compression scheme to select heavily depends on the nature of the expected queries. If we are always interested in all items of a query then the run-length bit vector scheme may be a clear winner.

Let us take a look at the cdfs shown in figure 5.2. For a given interval on the x-axis we can determine the percentage of queries that were answered in a time within our predefined interval. The run-length bit vector scheme is the orange line whereas the simple integer array is the black line. We can see that for about 95 percent of all queries the simple integer array beats all compression schemes. At the end however are the queries with an extremely large result and hence the compression and data representation of the run-length bit vector pays off. If we are only interested in the 95 percent then other options may be of interest. The best compression scheme only needs 25 GiB and is still very close. This however comes at a cost at run-time since it produces an uncompressed index as a result if a set operation involves indexes of different compression types. The regression line index is also close but compresses poorly and still needs 87 GiB of storage space. If we are interested in a good compression ratio for storage and runtime then the FoR index is a good candidate.

If the upper 5 percent of the queries are important as well then the run-length delta index is the best choice. It also features the best cell-computation time of all schemes while having a memory footprint close to the best variant. We use this index type as our default compression scheme due to its balanced trade-offs. Additionally, at least in the case of our web based demonstrator, retrieving all items of a result is not of great interest. Instead only small parts of a result are needed and often involve multiple cell operations.

Note that if we were interested in ranking the result then we could do so in

	Size [GiB]		Cells [s]		Ids [s]	Items [s]	
	Global	Local	Global	Local	Local	Global	Local
Array	170.9	158.0	329	365	65	1588	1497
Regression line	127.1	87.4	481	525	79	3527	3497
Rl bit vector	100.1	57.5	663	716	139	1180	1142
Rl delta	51.6	31.7	281	314	84	2183	2448
EliasFano	69.7	32.5	596	643	130	4590	4565
FoR	51.3	29.8	519	556	171	2532	2511
Patched FoR	44.2	25.5	590	691	226	3813	3737
Best	43.5	24.6	411	473	264	3943	3920

Table 5.9: Comparison of completion performance and index sizes for various index formats based on the web text queries. The "Cells" column lists the total time to compute the list of cells containing items. The "Id" column shows the time to convert local ids to global ids. Finally the "Items" column lists the time to compute all items of the given queries. All data is loaded completely into memory prior querying the data base. See section 5.5 for an in-depth discussion.

parallel for each cell and merge the result later. Incidentally this is what happens in our demonstrator on a lower zoom level where the top-k items of a set of cells is determined by considering only the top-k items of each cell.

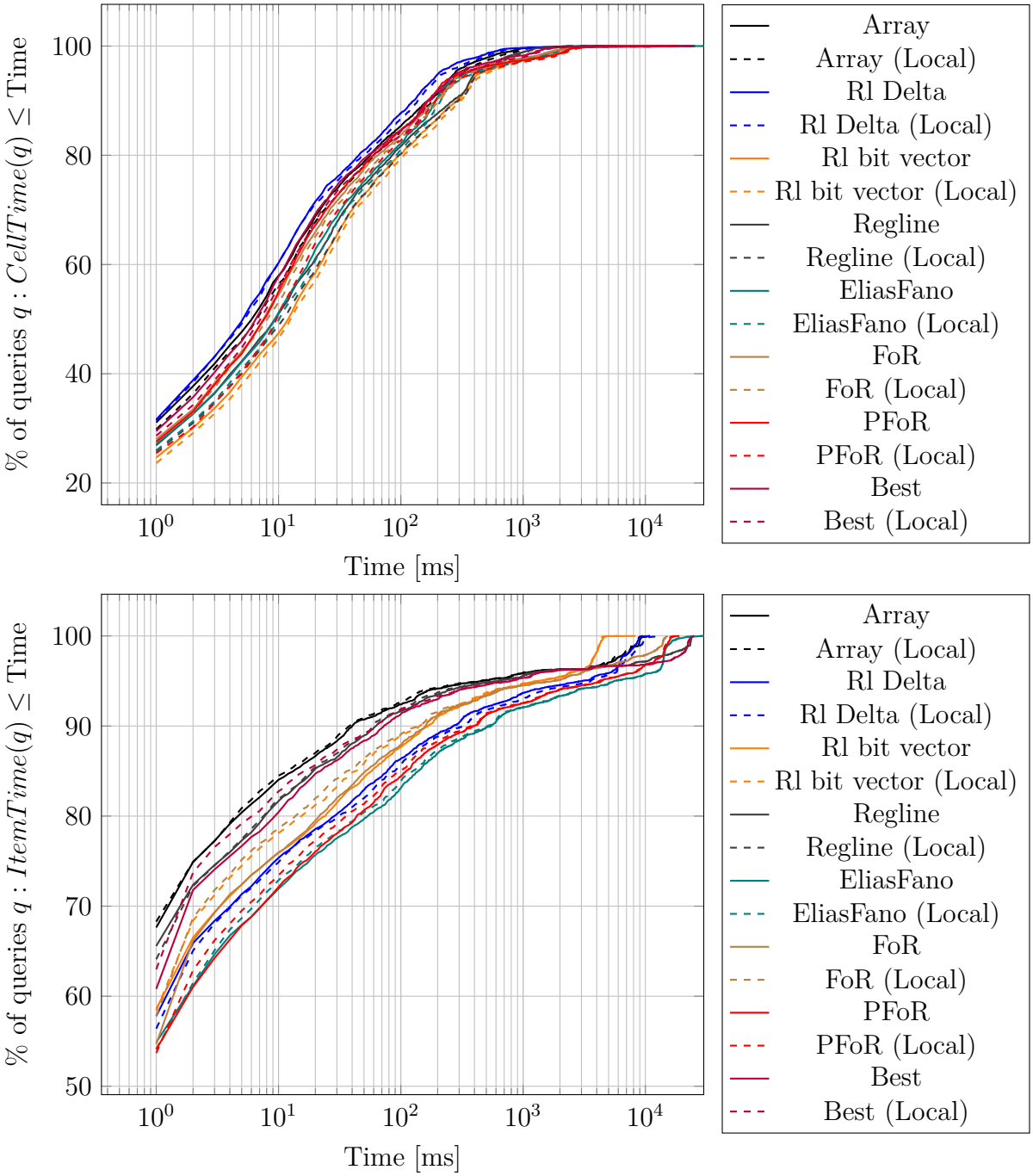


Figure 5.2: Cumulative distribution function of cell (top) and items (bottom) computation times of the website queries for all supported list encoding schemes for the Planet data set. Local denotes an index with cell local ids. See section 5.5 for an in-depth discussion.

5.6 Query Performance on Servers

In this section we would like to show the performance of OSCAR on powerful servers. We first describe how the size of the data influences the query timings. We then analyze the impact of the various cell refinement options. Finally we show how the delayed cell operation computation improves the query times both by optimizing cell operations and by the usage of multiple threads.

5.6.1 Performance scaling

Tables 5.10 and 5.11 show the mean and median time to compute the result of all textual queries of our test data set. Tables 5.12 and 5.13 almost look the same but this time only text queries containing at least one set operation were considered. If we compare these values with the characteristics of the respective data sets then we find that the query times increase linearly with the number of cells and items as we can also infer from figure 5.3. This matches with our expectation since most query operations are linear in the number of cells and items.

5.6.2 Impact of Cell Arrangement Refinements

In order to analyze the impact of the refinement and the cell arrangement in general we have to turn to tables 5.10, 5.12 and 5.14. The naive solution is a plain inverted index which always returns all matching items. Its strength are single query statement queries without a set operation and of course the request to return all matching items. If no set operation is involved then this instantly returns the result, hence the better completion times in table 5.10. However as soon as at least one set operation needs to be computed things change and mean and median times are worse than our cell arrangement based index as can be seen in table 5.12. Spatial queries are of course out of scope as well as efficient computation of the Inclusion-DAG.

What about the impact of the refinement? In theory we should get worse completion times for a cell arrangement with more cells. This is exactly what we can see in the tables mentioned before. The effect is especially pronounced for spatial queries (table 5.14): The connected cell arrangement has a mean cell completion time of 858 ms whereas the triangle refined one takes 2165 ms, more than twice the time. However the accuracy of some spatial queries like the ranged cell dilation operation increases. The time to compute the subgraph of the Inclusion-DAG and all items is not that much affected (also see figures 5.5, 5.6, 5.8 and 5.9).

Finally consider the cumulative distribution function of the cell completion time shown in figures 5.4 and 5.7. Here we can see that the effect of the refinement is evenly distributed among the queries. Interestingly the disconnected cell arrange-

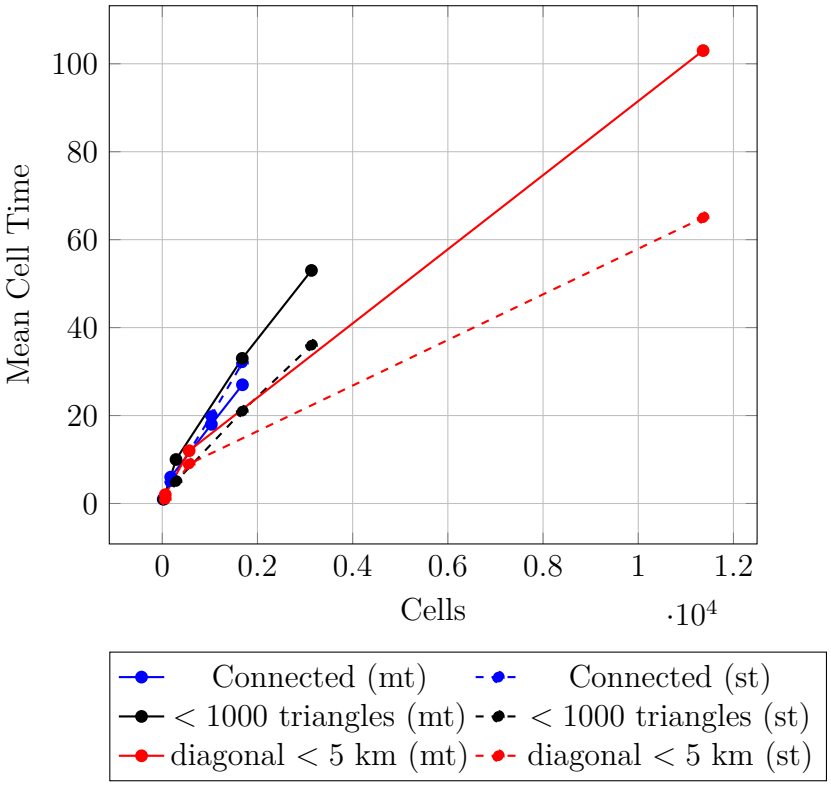


Figure 5.3: Mean cell completion times of the website text queries over the number of cells for various cell refinement types. st: single threaded, mt: multi-threaded with delayed cell computation.

	Ids	Ba-Wü	Germany	Europe	Planet
			Naive		
Items (μ/m) [ms]	Global	4/0	21/2	95/5	173/9
			Disconnected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	6/1	23/4	34/6
	Local	1/0	8/2	27/5	38/6
Id Mapping (μ/m) [ms]	Local	2/2	4/2	10/2	14/2
Subgraph (μ/m) [ms]	Global	2/0	14/0	58/0	83/0
	Local	2/0	13/0	56/0	78/0
Items (μ/m) [ms]	Global	10/1	50/1	220/2	356/2
	Local	10/1	49/1	216/2	348/2
			Connected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	6/1	18/3	27/4
	Local	1/0	7/2	22/4	32/5
Id Mapping (μ/m) [ms]	Local	2/2	4/2	10/2	15/2
Subgraph (μ/m) [ms]	Global	2/0	13/0	55/0	77/0
	Local	2/0	13/0	54/0	77/0
Items (μ/m) [ms]	Global	8/1	43/2	190/2	312/2
	Local	8/1	44/1	187/2	301/2
			Max 1000 triangles per cell		
Cells (μ/m) [ms]	Global	1/0	10/2	33/5	53/7
	Local	2/1	11/2	36/5	57/8
Id Mapping (μ/m) [ms]	Local	2/2	5/2	11/2	15/2
Subgraph (μ/m) [ms]	Global	2/0	16/0	70/0	97/0
	Local	2/0	16/0	67/0	99/0
Items (μ/m) [ms]	Global	11/1	50/2	227/2	369/2
	Local	10/1	49/2	222/2	365/2
			Max 5000m cell diagonal		
Cells (μ/m) [ms]	Global	2/0	12/2	103/5	−/−
	Local	2/0	14/2	112/6	−/−
Id Mapping (μ/m) [ms]	Local	2/2	5/2	29/2	−/−
Subgraph (μ/m) [ms]	Global	3/0	20/0	125/0	−/−
	Local	2/0	19/0	123/0	−/−
Items (μ/m) [ms]	Global	9/1	44/2	256/2	−/−
	Local	8/1	43/1	243/2	−/−

Table 5.10: Mean and median times to compute the result of queries for the website text queries using multiple threads. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. See section 5.6.1 for an in-depth discussion.

	Ids	Ba-Wü	Germany	Europe	Planet
			Naive		
Items (μ/m) [ms]	Global	4/0	21/2	95/5	173/9
			Disconnected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	4/1	18/1	28/2
	Local	1/0	4/1	14/2	22/2
Id Mapping (μ/m) [ms]	Local	5/0	24/0	82/1	135/1
Subgraph (μ/m) [ms]	Global	1/0	9/0	49/0	70/0
	Local	1/0	9/0	48/0	70/0
Items (μ/m) [ms]	Global	10/0	104/0	642/0	989/0
	Local	10/0	101/0	627/0	993/0
			Connected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	5/0	20/1	32/2
	Local	1/0	4/1	16/2	25/2
Id Mapping (μ/m) [ms]	Local	5/0	25/0	84/1	137/1
Subgraph (μ/m) [ms]	Global	1/0	9/0	47/0	69/0
	Local	1/0	9/0	49/0	69/0
Items (μ/m) [ms]	Global	10/0	101/0	609/0	948/0
	Local	10/0	100/0	608/0	930/0
			Max 1000 triangles per cell		
Cells (μ/m) [ms]	Global	1/0	5/1	21/2	36/3
	Local	1/0	5/1	19/2	30/3
Id Mapping (μ/m) [ms]	Local	4/0	24/0	79/0	123/0
Subgraph (μ/m) [ms]	Global	1/0	11/0	58/0	87/0
	Local	1/0	12/0	59/0	86/0
Items (μ/m) [ms]	Global	12/0	119/0	711/0	1115/0
	Local	12/0	118/0	726/0	1099/0
			Max 5000m cell diagonal		
Cells (μ/m) [ms]	Global	1/0	9/1	65/3	−/−
	Local	1/0	9/1	67/3	−/−
Id Mapping (μ/m) [ms]	Local	5/0	26/0	103/0	−/−
Subgraph (μ/m) [ms]	Global	1/0	14/0	115/0	−/−
	Local	1/0	15/0	113/0	−/−
Items (μ/m) [ms]	Global	13/0	127/0	1114/0	−/−
	Local	13/0	125/0	1100/0	−/−

Table 5.11: Mean and median times to compute the result of queries for the website text queries using a single thread. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. See section 5.6.1 for an in-depth discussion.

	Ids	Ba-Wü	Germany	Europe	Planet
			Naive		
Items (μ/m) [ms]	Global	7/3	34/12	150/43	265/48
			Disconnected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	5/1	20/5	29/7
	Local	1/0	7/2	22/5	31/8
Id Mapping (μ/m) [ms]	Local	2/1	3/2	5/2	6/2
Subgraph (μ/m) [ms]	Global	1/0	5/0	16/0	21/0
	Local	1/0	5/0	16/0	20/0
Items (μ/m) [ms]	Global	4/0	18/1	62/1	88/1
	Local	4/0	18/1	60/1	86/1
			Connected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	6/1	19/4	28/5
	Local	1/0	7/1	20/4	29/5
Id Mapping (μ/m) [ms]	Local	2/1	3/2	5/2	6/2
Subgraph (μ/m) [ms]	Global	1/0	5/0	16/0	21/0
	Local	1/0	5/0	15/0	20/0
Items (μ/m) [ms]	Global	3/0	17/1	55/1	80/1
	Local	3/0	16/1	57/1	78/1
			Max 1000 triangles per cell		
Cells (μ/m) [ms]	Global	1/1	9/2	27/6	43/9
	Local	1/1	9/2	30/7	46/10
Id Mapping (μ/m) [ms]	Local	2/1	3/2	5/2	6/2
Subgraph (μ/m) [ms]	Global	1/0	6/0	19/0	25/0
	Local	1/0	6/0	18/0	25/0
Items (μ/m) [ms]	Global	4/0	18/1	64/1	92/1
	Local	4/0	18/1	62/1	91/1
			Max 5000m cell diagonal		
Cells (μ/m) [ms]	Global	2/1	13/2	92/7	−/−
	Local	2/1	14/2	96/8	−/−
Id Mapping (μ/m) [ms]	Local	2/1	3/2	10/2	−/−
Subgraph (μ/m) [ms]	Global	1/0	7/0	32/0	−/−
	Local	1/0	7/0	32/0	−/−
Items (μ/m) [ms]	Global	3/0	16/1	70/1	−/−
	Local	3/0	16/1	71/1	−/−

Table 5.12: Mean and median times to compute the result of queries for the website text queries containing at least one set operation using multiple threads. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. See section 5.6.1 for an in-depth discussion.

	Ids	Ba-Wü	Germany	Europe	Planet
			Naive		
Items (μ/m) [ms]	Global	7/3	34/12	150/43	265/48
			Disconnected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	6/1	25/3	37/4
	Local	1/0	5/1	17/3	26/4
Id Mapping (μ/m) [ms]	Local	2/0	14/0	34/0	52/0
Subgraph (μ/m) [ms]	Global	0/0	3/0	14/0	18/0
	Local	0/0	3/0	13/0	18/0
Items (μ/m) [ms]	Global	4/0	40/0	221/0	278/0
	Local	4/0	39/0	218/0	275/0
			Connected Cell Arrangement		
Cells (μ/m) [ms]	Global	1/0	7/1	27/3	43/4
	Local	1/0	6/1	21/3	31/4
Id Mapping (μ/m) [ms]	Local	2/0	14/0	35/0	52/0
Subgraph (μ/m) [ms]	Global	0/0	3/0	13/0	18/0
	Local	0/0	3/0	14/0	18/0
Items (μ/m) [ms]	Global	4/0	40/0	212/0	268/0
	Local	4/0	40/0	212/0	268/0
			Max 1000 triangles per cell		
Cells (μ/m) [ms]	Global	1/0	7/1	29/3	47/5
	Local	1/0	6/1	22/3	36/5
Id Mapping (μ/m) [ms]	Local	2/0	13/0	33/0	48/0
Subgraph (μ/m) [ms]	Global	0/0	4/0	16/0	22/0
	Local	0/0	4/0	16/0	22/0
Items (μ/m) [ms]	Global	4/0	46/0	249/0	325/0
	Local	4/0	44/0	251/0	325/0
			Max 5000m cell diagonal		
Cells (μ/m) [ms]	Global	2/0	13/1	78/5	−/−
	Local	1/0	11/1	72/5	−/−
Id Mapping (μ/m) [ms]	Local	2/0	15/0	39/0	−/−
Subgraph (μ/m) [ms]	Global	1/0	5/0	30/0	−/−
	Local	1/0	5/0	29/0	−/−
Items (μ/m) [ms]	Global	5/0	48/0	352/0	−/−
	Local	5/0	47/0	349/0	−/−

Table 5.13: Mean and median times to compute the result of queries for the website text queries containing at least one set operation using a single thread. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. See section 5.6.1 for an in-depth discussion.

	Ids	Ba-Wü	Germany	Europe	Planet
		Disconnected Cell Arrangement			
Cells (μ/m) [ms]	Global	26/3	304/9	1228/23	1342/35
	Local	10/2	179/6	905/10	806/15
Id Mapping (μ/m) [ms]	Local	2/2	2/2	4/2	4/2
Subgraph (μ/m) [ms]	Global	1/0	5/0	16/0	21/0
	Local	1/0	5/0	15/0	20/0
Items (μ/m) [ms]	Global	11/1	29/1	71/1	100/1
	Local	10/1	27/1	64/1	90/1
		Connected Cell Arrangement			
Cells (μ/m) [ms]	Global	10/3	173/5	836/9	858/12
	Local	10/2	179/6	834/10	806/15
Id Mapping (μ/m) [ms]	Local	2/2	2/2	4/2	4/2
Subgraph (μ/m) [ms]	Global	1/0	5/0	15/0	20/0
	Local	1/0	5/0	16/0	20/0
Items (μ/m) [ms]	Global	10/1	27/1	61/1	90/1
	Local	10/1	27/1	62/1	88/1
		Max 1000 triangles per cell			
Cells (μ/m) [ms]	Global	25/4	496/8	2223/17	2165/25
	Local	17/3	458/7	2089/11	2138/19
Id Mapping (μ/m) [ms]	Local	2/2	3/2	4/2	5/2
Subgraph (μ/m) [ms]	Global	2/0	7/0	19/0	25/0
	Local	2/0	6/0	18/0	24/0
Items (μ/m) [ms]	Global	11/1	29/1	71/1	104/1
	Local	10/1	27/1	63/1	96/1
		Max 5000m cell diagonal			
Cells (μ/m) [ms]	Global	36/4	1271/7	11224/16	−/−
	Local	35/3	1280/8	11189/19	−/−
Id Mapping (μ/m) [ms]	Local	2/2	3/2	12/2	−/−
Subgraph (μ/m) [ms]	Global	2/0	8/0	31/0	−/−
	Local	2/0	7/0	30/0	−/−
Items (μ/m) [ms]	Global	10/1	29/1	72/1	−/−
	Local	10/1	28/1	68/1	−/−

Table 5.14: Mean and median times to compute the result of queries for the website spatial queries using multiple threads. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. The cell computation time is rather high since the spatial predicates are not fully optimized yet. See section 5.6.1 for an in-depth discussion.

	Ids	Ba-Wü	Germany	Europe	Planet
		Disconnected Cell Arrangement			
Cells (μ/m) [ms]	Global	83/1	2542/4	14544/8	13532/12
	Local	79/1	2387/3	14068/7	12846/10
Id Mapping (μ/m) [ms]	Local	1/0	6/0	17/0	22/0
Subgraph (μ/m) [ms]	Global	1/0	3/0	13/0	18/0
	Local	1/0	3/0	13/0	17/0
Items (μ/m) [ms]	Global	18/0	77/0	238/0	311/0
	Local	17/0	73/0	219/0	301/0
		Connected Cell Arrangement			
Cells (μ/m) [ms]	Global	75/1	2436/3	13743/7	13010/10
	Local	76/1	2430/3	13561/7	13103/10
Id Mapping (μ/m) [ms]	Local	1/0	6/0	17/0	21/0
Subgraph (μ/m) [ms]	Global	1/0	3/0	13/0	18/0
	Local	1/0	3/0	13/0	17/0
Items (μ/m) [ms]	Global	17/0	75/0	217/0	289/0
	Local	17/0	74/0	217/0	288/0
		Max 1000 triangles per cell			
Cells (μ/m) [ms]	Global	212/2	7570/5	40486/9	40419/14
	Local	199/1	7129/4	38232/8	38297/13
Id Mapping (μ/m) [ms]	Local	1/0	6/0	15/0	19/0
Subgraph (μ/m) [ms]	Global	1/0	4/0	16/0	22/0
	Local	1/0	4/0	14/0	21/0
Items (μ/m) [ms]	Global	19/0	95/0	263/0	345/0
	Local	17/0	89/0	235/0	324/0
		Max 5000m cell diagonal			
Cells (μ/m) [ms]	Global	542/1	24728/4	270452/12	−/−
	Local	529/1	25233/4	268144/12	−/−
Id Mapping (μ/m) [ms]	Local	1/0	6/0	17/0	−/−
Subgraph (μ/m) [ms]	Global	1/0	5/0	28/0	−/−
	Local	1/0	5/0	28/0	−/−
Items (μ/m) [ms]	Global	19/0	119/0	283/0	−/−
	Local	19/0	119/0	278/0	−/−

Table 5.15: Mean and median times to compute the result of queries for the website spatial queries using a single thread. The Cells row gives the time to compute all relevant cells of a query. The Subgraph row gives the time to compute the subgraph of the Inclusion-Dag. The Map row has the time to map local ids to global ids. Finally the Items row lists the time to compute all items of a result. The cell computation time is rather high since the spatial predicates are not fully optimized yet. See section 5.6.1 for an in-depth discussion.

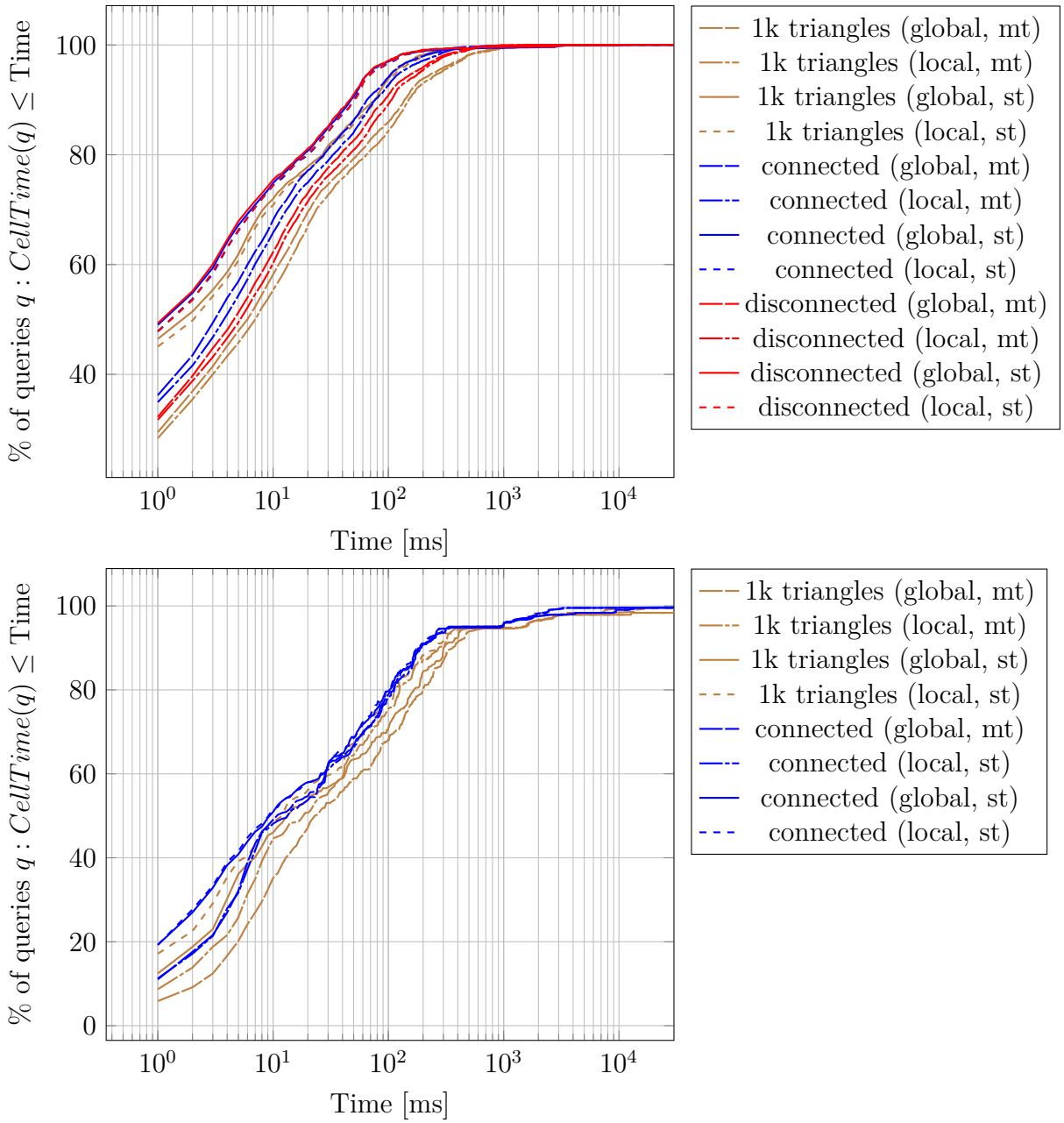


Figure 5.4: Cumulative distribution function to compute the cells of a query showing the impact of cell refinement and local vs. global item ids for the Planet data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

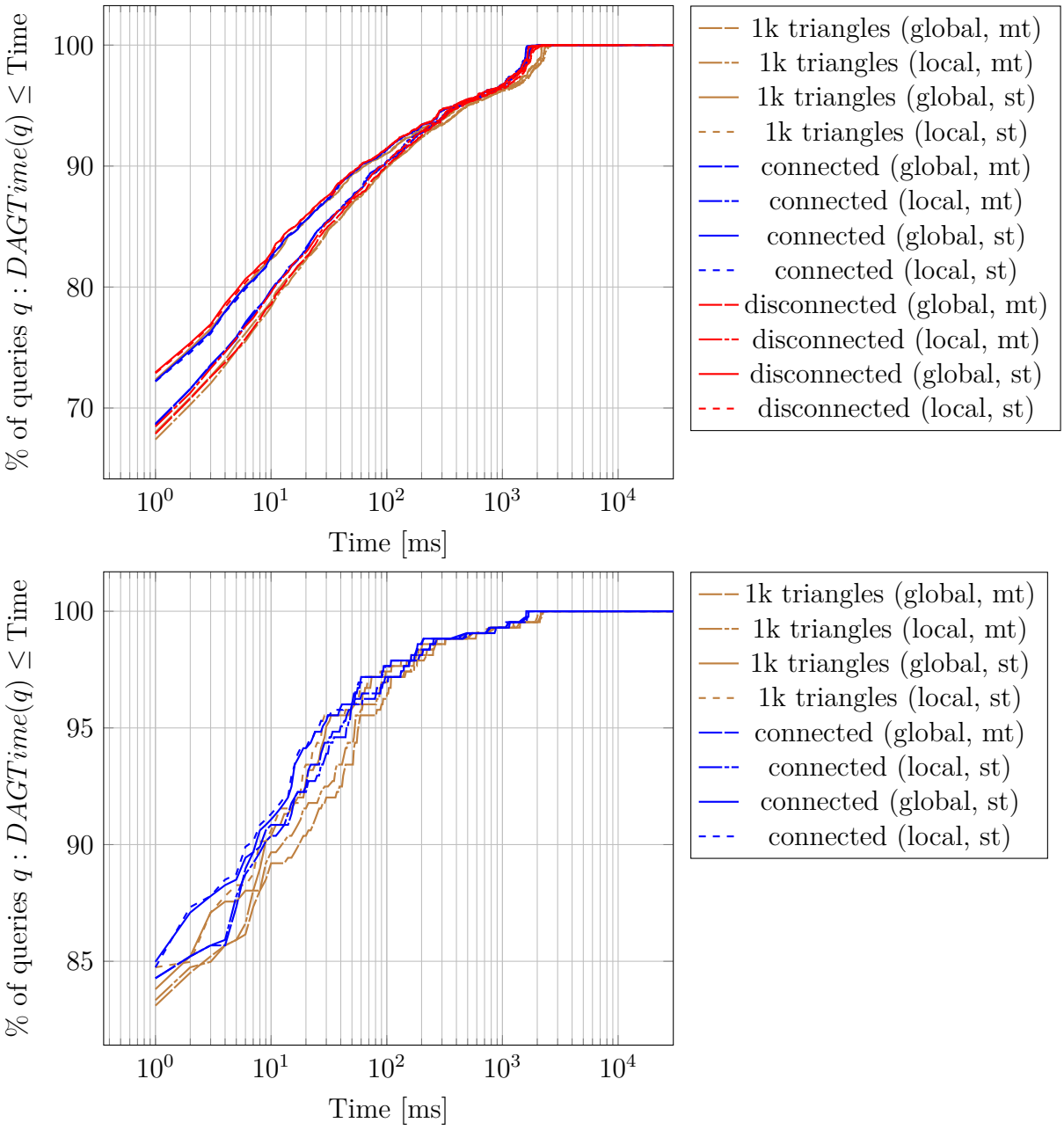


Figure 5.5: Cumulative distribution function to compute the subgraph of a query showing the impact of cell refinement and local vs. global item ids for the Planet data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

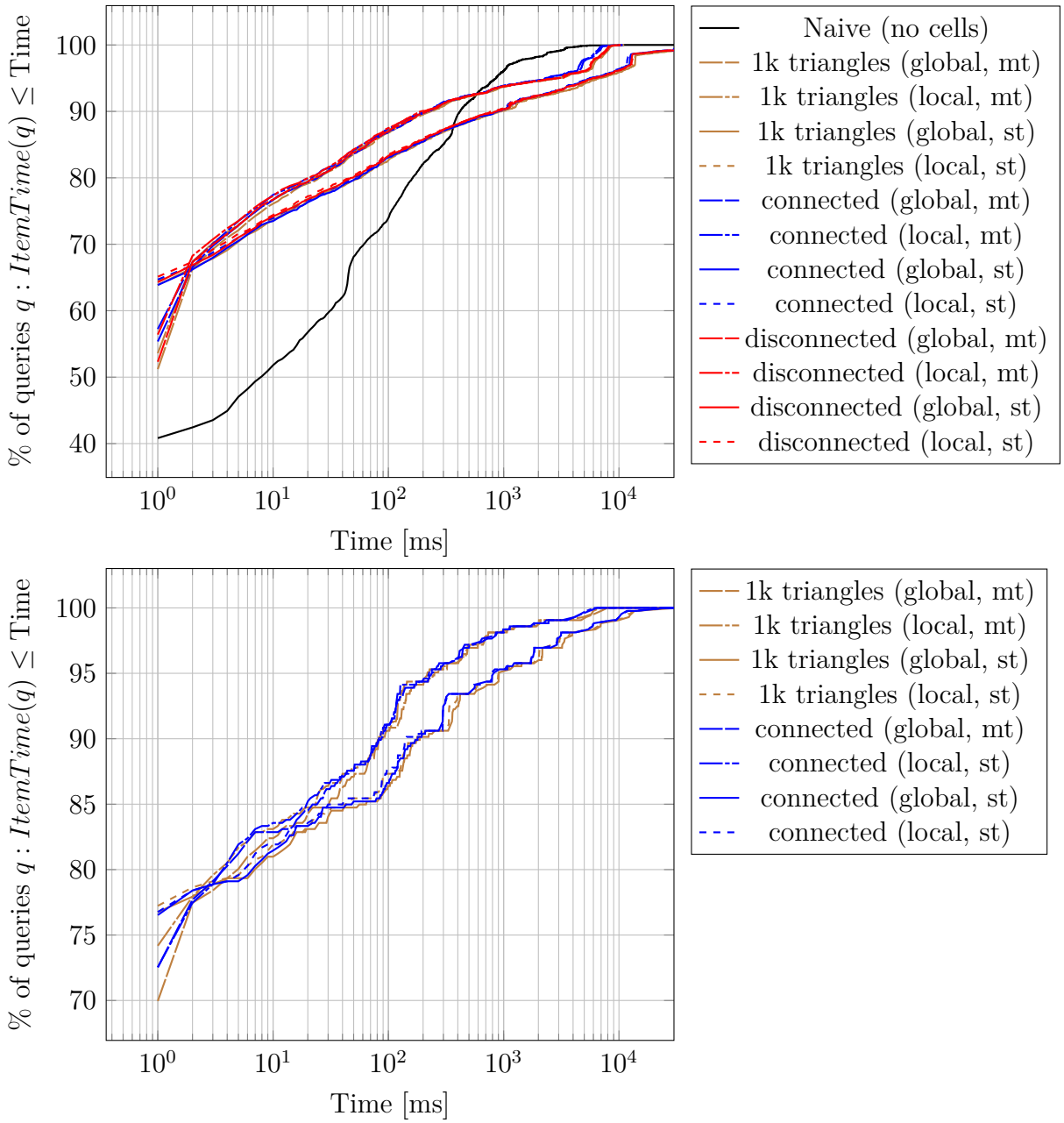


Figure 5.6: Cumulative distribution function to compute all items of a query showing the impact of cell refinement and local vs. global item ids for the Planet data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

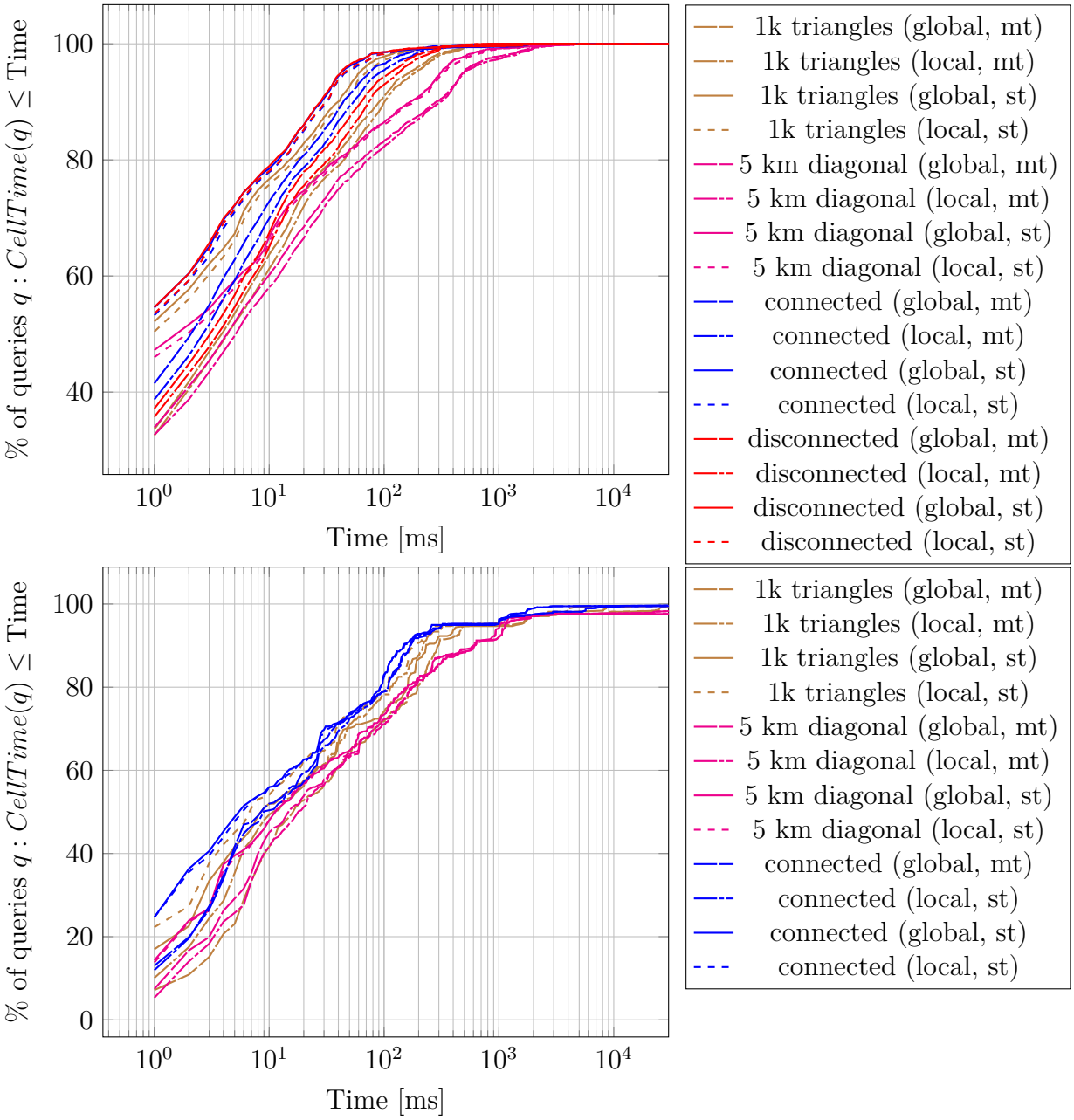


Figure 5.7: Cumulative distribution function to compute the cells of a query showing the impact of cell refinement and local vs. global item ids for the Europe data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

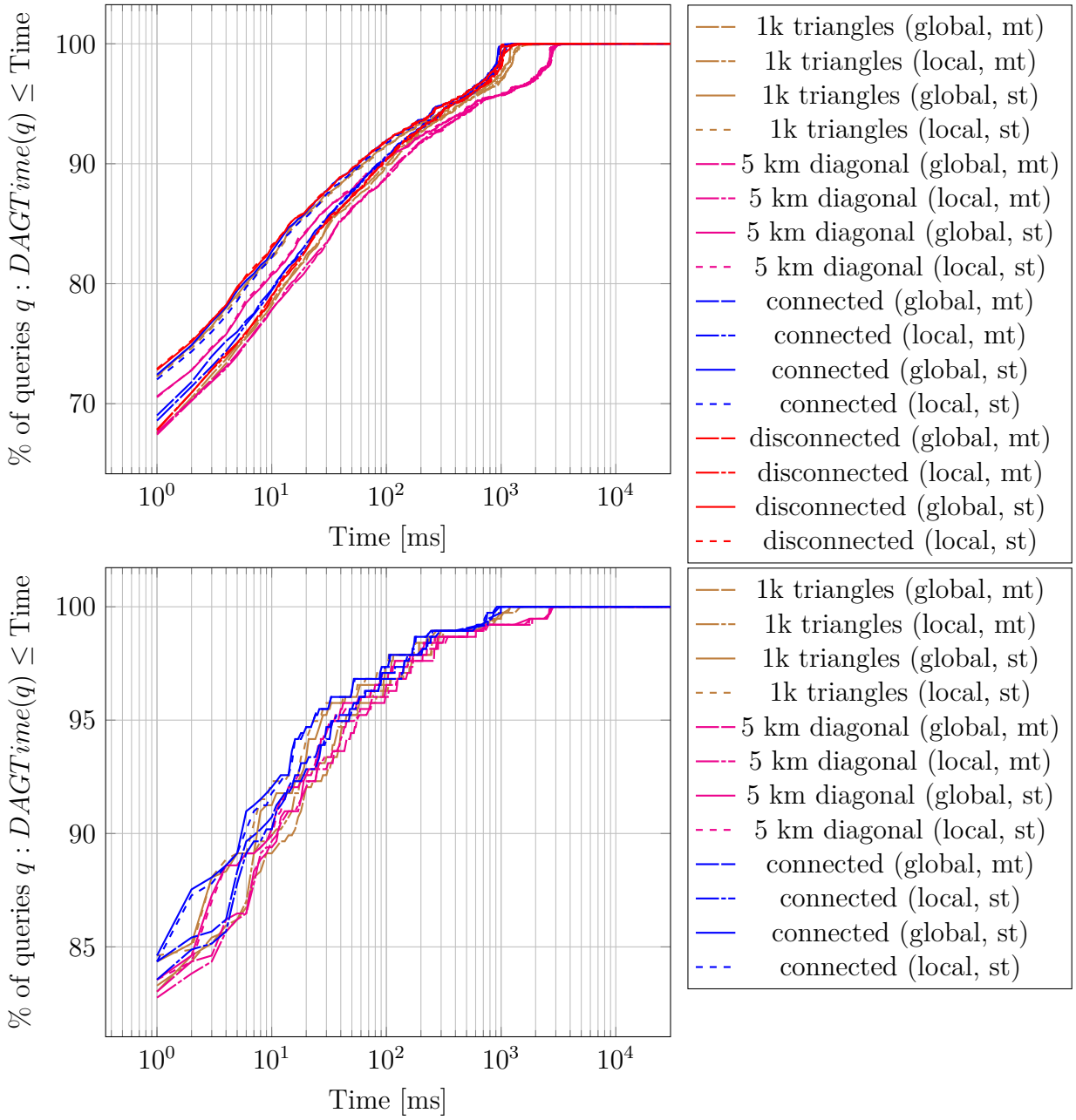


Figure 5.8: Cumulative distribution function to compute the subgraph of a query showing the impact of cell refinement and local vs. global item ids for the Europe data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

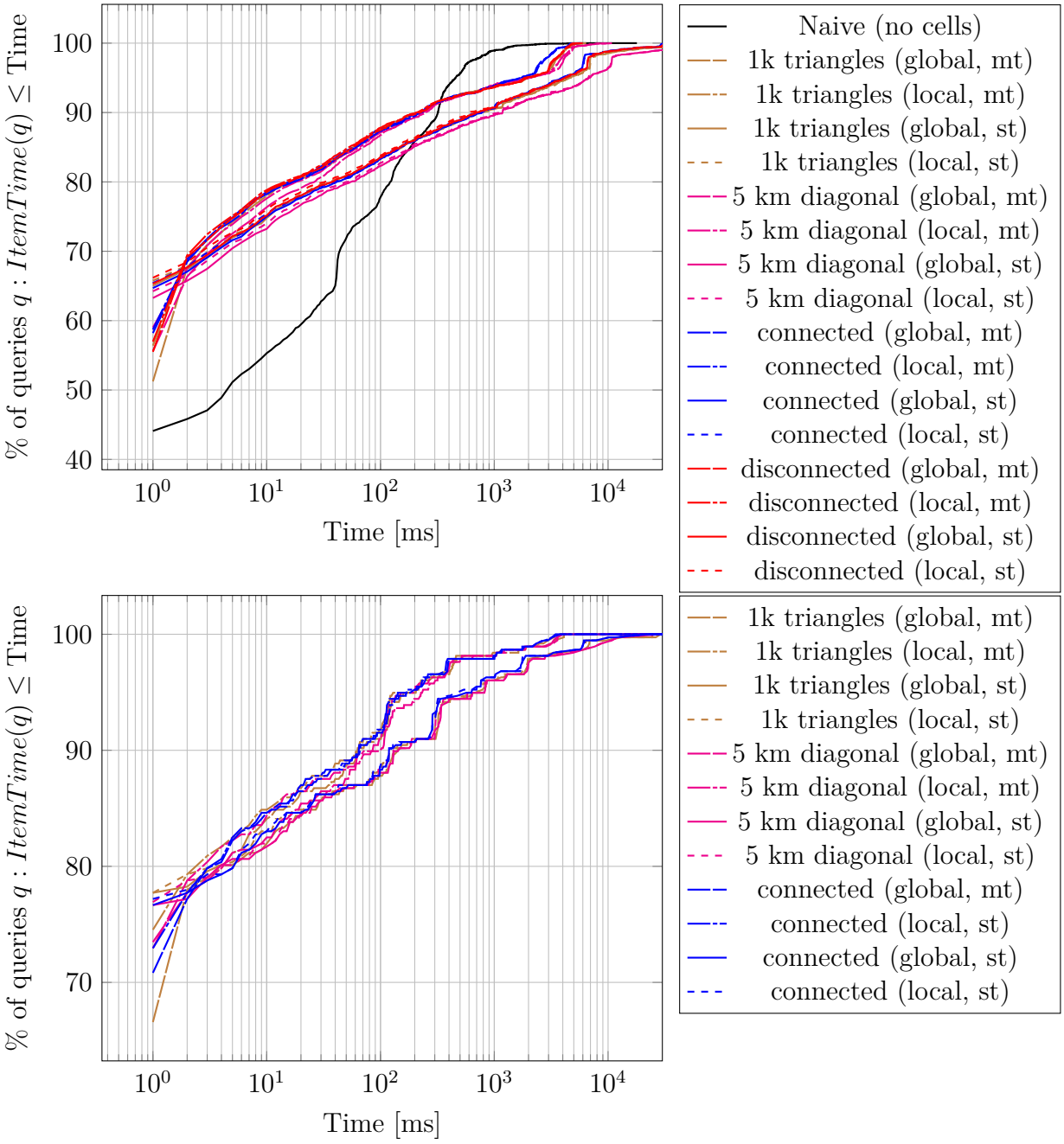


Figure 5.9: Cumulative distribution function to compute all items of a query showing the impact of cell refinement and local vs. global item ids for the Europe data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

Threads [1]	1	2	4	8	12	24	48
Set Operations [ms]	2850	1550	800	460	330	185	150
Speed-up	1	1.8	3.5	6.1	8.6	15.4	19
Efficiency	1	0.9	0.9	0.8	0.7	0.6	0.4

Table 5.16: Efficiency ratios for the query "*@highway @building @amenity*" using multiple threads. Note that our benchmark system is a dual-socket machine with 12 real cores per CPU. This likely contributes to the drop in efficiency with increasing thread count due to synchronization of global locks.

ment is slower than connected cell arrangement. This should come as a surprise since it has almost the same number of cells. It however contains larger cells that likely contribute to higher completion times.

5.6.3 Delayed Computation

The main difference between the CellQueryResult and the DCQR is the execution of set operations. Using the DCQR these operations are delayed until the user explicitly decides to execute these. If a query removes many cells like the "*@highway @amenity @building #"Stuttgart""* query then delaying the execution of the cell set operations drastically improves the computation time since almost all intersections can be decided on the cell level by the last token (CQR: 3100 ms, DCQR 300 ms). On the other hand the CellQueryResult is more efficient in computing the result of the query "*@highway @amenity @building*" since now all partial-partial-match set operations need to be executed and almost none can be decided on the cell level (CQR: 3100 ms, DCQR 3200 ms).

Additionally the delayed computation scheme allows set operations to be computed in parallel to further improve the query times. Our implementation parallelizes the execution of the set operations and some parts of the reduction phase where we have to unite the results of each cell to retrieve all items. Computing the set operations should in theory scale linearly in the number of processors with a speed-up of 1. In practice we get an efficiency ratio of 0.4 to 0.9 for this query as depicted in table 5.16. The lower efficiency ratio in a high thread count setting is likely attributable to the fact that our machine has 24 real cores with a total of 48 logical cores. In the case of 48 worker threads this results in two threads sharing a single physical core thus reducing the efficiency of the parallelization.

It may come as a surprise that the significantly lower processing times achievable do not translate to lower processing times of our benchmark query data set as can be seen in figures 5.4 and 5.7. This is likely caused by the fact that queries with

cell counts below a hundred thousand cells do not benefit very much from multi-threading. To some degree this is the fault of our implementation: we currently do not have a central thread manager to keep threads alive. Instead we spawn and kill threads in each parallelized step of our execution pipeline. On our server with 48 threads this boils down to at least one millisecond per step. This has a considerable impact on the cell computation time especially if the query is rather small and the first thread likely computes the result on its own. Additionally the delayed computation scheme is only useful if it can prune many cells due to a region constraint. If a query has many partial-match set operations then these have to be computed in any case but additionally the operation trees have to be created as well. In case of the planet data set only a small number of queries benefit from multiple threads in conjunction with the delayed processing scheme as we can infer from figures 5.10 and 5.11. The item computations on the other hand benefit from multiple threads.

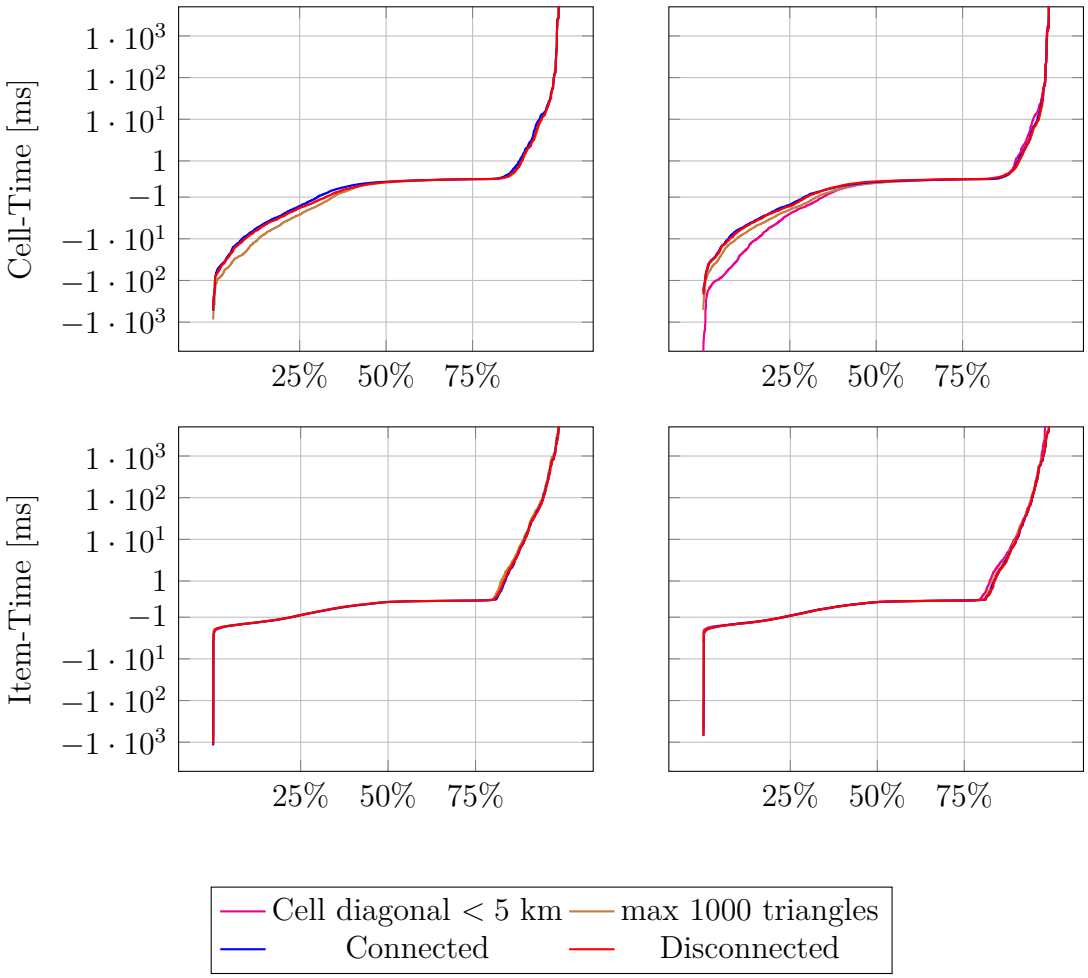


Figure 5.10: Cumulative distribution function of the difference between the single threaded (CQR) and the multi-threaded delayed computation scheme (DCQ). In short we compute *cdf* $(\mathrm{CQR}_t(q) - \mathrm{DCQR}_t(q))$ over all queries q. Depicted are the website text queries containing set operations with the cell time at the top and item time at the bottom. The left row shows the planet data set, the right row shows the Europe data set.

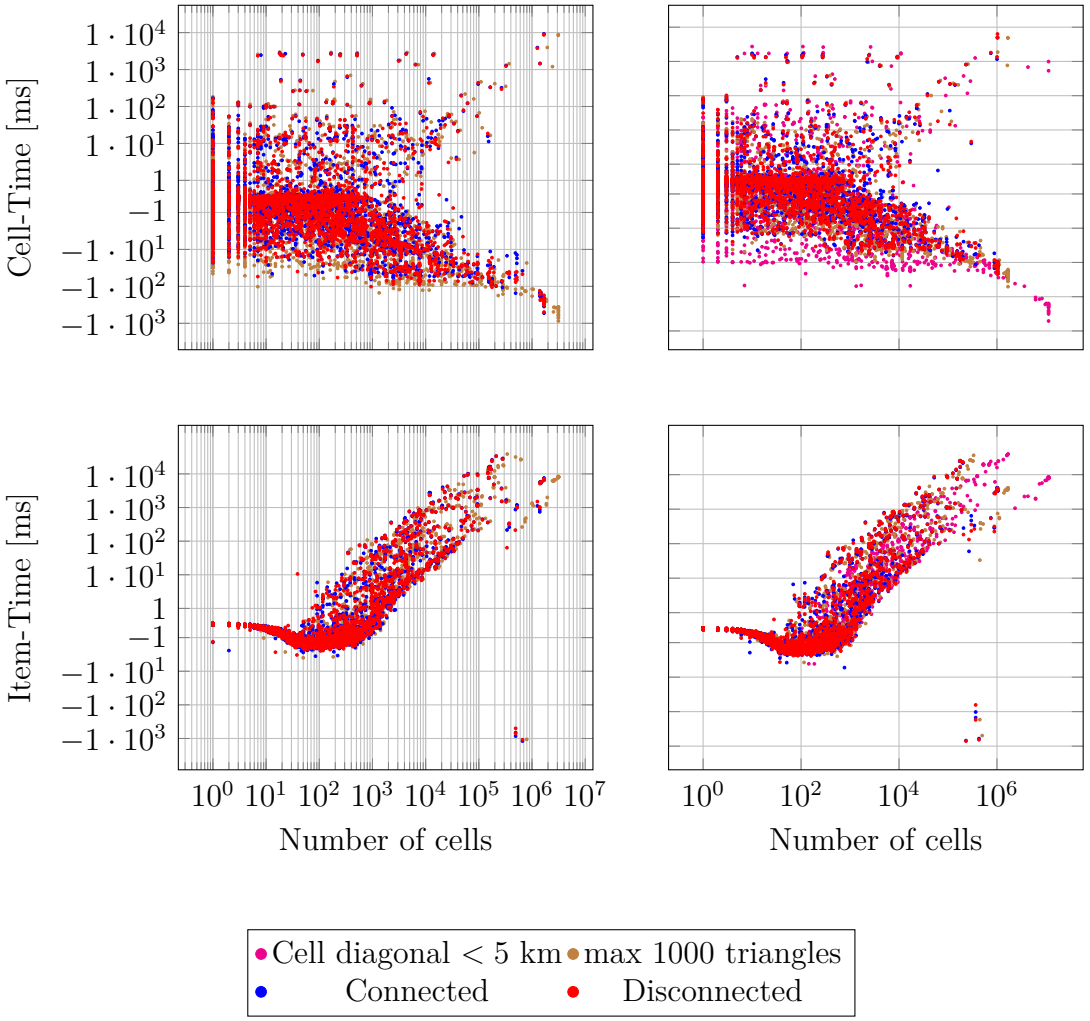


Figure 5.11: Scatter plot of the difference between the single threaded and the multi-threaded delayed computation scheme. Depicted are the website text queries containing set operations with the cell time at the top and item time at the bottom. The left row shows the planet data set, the right row shows the Europe data set.

5.7 Mobile Phone Benchmarks

We conducted some benchmarks on a Sony Xperia V running Android 4.4 to show OSCAR's capabilities to cope with low performance environments. Its specifications are listed in table 5.1.

Setup We installed a Debian Stretch chroot environment on the internal storage which allowed us to compile and run OSCAR on the mobile phone inside the chroot. Another option would be to compile CGAL and all of its dependencies with the Android NDK toolchain. This however is a very cumbersome task to do [92].

Results We used the Baden-Württemberg and Germany data since the Europe and Planet data set are too large to fit on a 32 GB SD card. Our mobile platform has only 1 GiB of memory of which about 200 MiB are used by the graphics system. It is therefore impossible to keep the whole index in memory. Instead most of the data is loaded on demand from the SD card. For the Baden-Württemberg data set it is still possible to use a memory mapped storage back end. The Germany data set however is too large to fit into a 32 bit address space. Instead we have to use a back end that loads data on request from the files without using memory maps. The file system buffer of the operating system is involved as well, but the access is not as efficient as the memory mapped variant. We also compare the impact of the file system cache by dropping all cached data before a query string is processed. In order to do this we write a 1 to `/proc/sys/vm/drop_caches` which drops the page cache. We use a single thread to process the requests with the CellQueryResult class. Figure 5.12 shows the cdf of the text queries with set operations and the cdf of the spatial queries. For the Baden-Württemberg data set most queries can be answered in less than a second even in the case of non-cached data. For the Germany data set this is only true if the data is already in memory. If all data has to be loaded then only about 80% of all queries are under the one second mark. There are even some queries, like "*@highway @amenity @building*", taking more than 60 seconds, though these are highly unlikely to appear on a mobile system.

Spatial queries on the other hand take longer with only 80 % under the one second mark.

Non-cached data has a pronounced impact on the query times at least for all regular queries. Hence being able to keep more data in memory is a very important feature of the compressed variants. The amount of data that needs to be fetched does not seem to play such an important role since the local item id index is always very close to the global item id index. This is likely caused by the file system cache which always loads about 128 KiB of data. The deduplication used to store the

posting lists results in a more random access pattern reducing the usefulness of the prefetching by the operating system. However reducing the prefetching amount does not improve the query timings but rather increases them indicating its benefit.

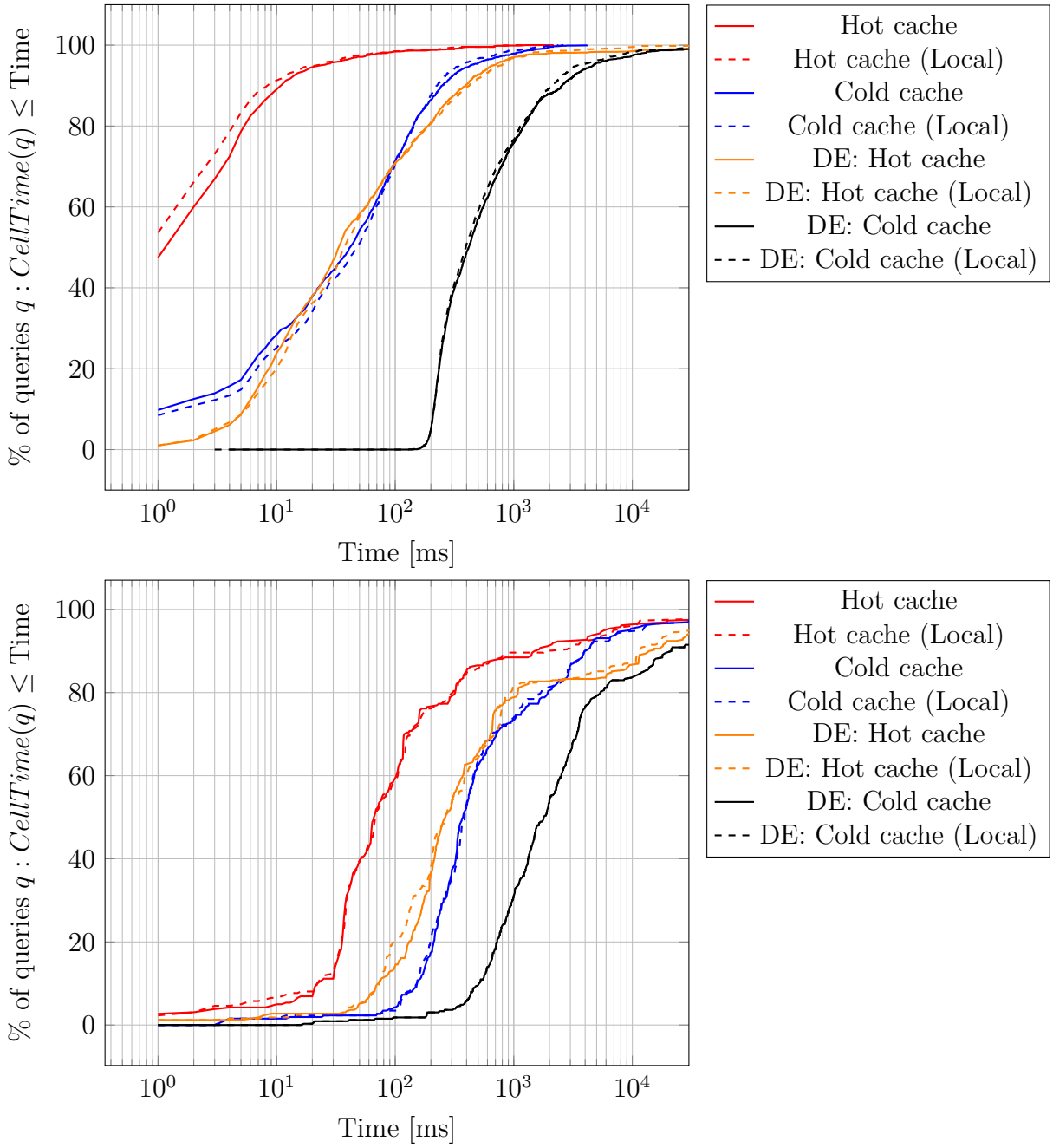


Figure 5.12: Cumulative distribution functions for global and local item ids and hot and cold caches measured on our mobile platform. The top picture shows the text queries containing at least one set operation. The bottom picture shows the spatial queries.

5.8 Alternative Map Tesselations

OSCAR uses a special kind of map tessellation induced by the regions of the data set with some additional refinement steps. This however produces cells that are unequal in size and form, hence the refinement steps. Unfortunately these do not result in a tessellation with cells of equal size, rather cells are somewhat similar.

There are however also many advantages to this representation. First of all it is easy to compute the subgraph of the Inclusion-DAG for a given set of cells. Secondly the cells returned for a given query exactly match the given query, no additional filtering is needed. Thirdly region queries always return full-match cells and item queries usually return partial match cells. This allows us to store the result for region and item queries in a single data stream. Depending on the used query we can either only return the full-match cells, the partial-match cells or combine both which is a fast operation.

We can get rid of the downsides by using more regular tessellations for example by using the discrete global grids (DGG) introduced in section 2.3.1. In turn we lose the aforementioned advantages, especially the semantic information of the subgraph of the Inclusion-DAG or its easy computation. If we are only interested in some kind of hierarchical representation then we can simply use the hierarchy provided by the spatial grid which we can again compute easily from a given result.

In the following we would like to show the effect of different spatial grids on the query times and storage space. To this end we implemented the text search part of OSCAR based on an abstract notion of a spatial grid. We first compute for each grid cell all of its covered items. We then transform all entries of our text search structures such that they are valid for our new map tessellation. Again we can use the notion of full-match and partial-match cells though we cannot assume that region queries only result in full-match cells. We therefore cannot store the necessary data as efficiently as possible with our original tessellation. Instead we store the entries for region, item and mixed queries in separate locations. In order to save space while increasing the query times we can opt to only store entries for region and item queries and compute the result of the mixed queries during query computation. This however is not as fast as in our original variant since region queries do not necessarily return only full-match cells.

5.8.1 Results

All of our tested grids are hierarchical in nature allowing a user to choose a level of accuracy. We chose the level of accuracy such that the cells of the induced cell arrangement have a size of about 40 square kilometers. See table 5.17 for an overview of various characteristics of the resulting data structures for the planet data set. Compared to the triangulation based cell arrangement the storage space

increases by a factor of less than two. Compared to the variant with small cells the difference is not that pronounced. Yet if we chose to produce smaller cells using the next refinement level the storage overhead increases from about 70 GiB for the H3 index to more than 240 GiB. Note the high maximum cell area of the triangulation based cell arrangement. This is mostly due to the fact that we don't refine triangles that are not covered by any region. Additionally our simple snapping algorithm may degrade the triangle refinement properties in some areas. For the arrangement with a cell diagonal constraint this results in cells being made up of only a single triangle. The other refinement types contain cells that are rather large. The impact of these large cells is likely very low since they account for less than 0.4 percent of all cells containing less than 2 percent of all items.

	Cell area min/mean/max $[km^2]$	Cells $[10^3]$	Index Size $[GiB]$	Search Size $[GiB]$
HTM	40/46/60	5536.4	87.5	80.4
H3	36/36/36	2709.0	74.1	69.4
Grid	0/35/47	2829.9	70.5	68.6
OSCAR				
disconnected	$0/303/18M$	1684.4	23.4	17.8
connected	$0/303/18M$	1684.4	43.1	33.1
max 1000 triangles	$0/117/6M$	3134.3	51.6	35.9
cell diagonal < 5 km	$0/1.3/140k$	50239.9	≈ 70	≈ 55

Table 5.17: Characteristics of the cell arrangement induced by alternative map tessellations of the planet data set. We could not compute the search structures for the planet data set and relied on an approximation based on the Europe data set which is always within a factor of two for other refinement settings. The high maximum cell area is in part caused by our snapping procedure (see section 4.3.2) and because we chose to not refine cells and triangles not covered by any region.

Figures 5.13 and 5.14 show the completion performance for the planet and Europe data set. On the planet data set all cell arrangements are very close together with no clear winner. This is not true anymore for the Europe data set where the H3 index and the simple grid dominate the other arrangements if we are only interested in cells. Item computation on the other hand is close together as well.

We can further analyze the difference between the arrangements using figures 5.15 and 5.16 which show the difference of the completion times of each query for every grid type. To be precise for cell arrangements A and B and query q we compute $A_t(q) - B_t(q)$ where $X_t(q)$ indicates the time it takes to process the

query q using arrangement X. We then plot the cumulative distribution function of these values.

We again find that there is no clear winner which dominates every other arrangement type. In general we can say that in about a third of the queries one arrangement is faster than the other and in about a third they are on par. Table 5.18 lists the number of queries for each grid type where the respective grid computed the query result in the least amount of time. In all data sets we find that the HTM index is the slowest of all variants. The simple grid and the H3 index are almost on par, with the H3 index being slightly faster. The performance of the triangulation based cell arrangement heavily depends on the used refinement type. Naturally the arrangement with the most number of cells looses to all other arrangements. Overall the H3 index seems to be the most promising candidate for an alternative map tessellation. Compared to the HTM index and the simple grid the H3 index has the disadvantage that its hierarchy does not allow a refinement of cells, but rather each level has its own cell arrangement. This may pose a problem for the hierarchical variant proposed in section 7.

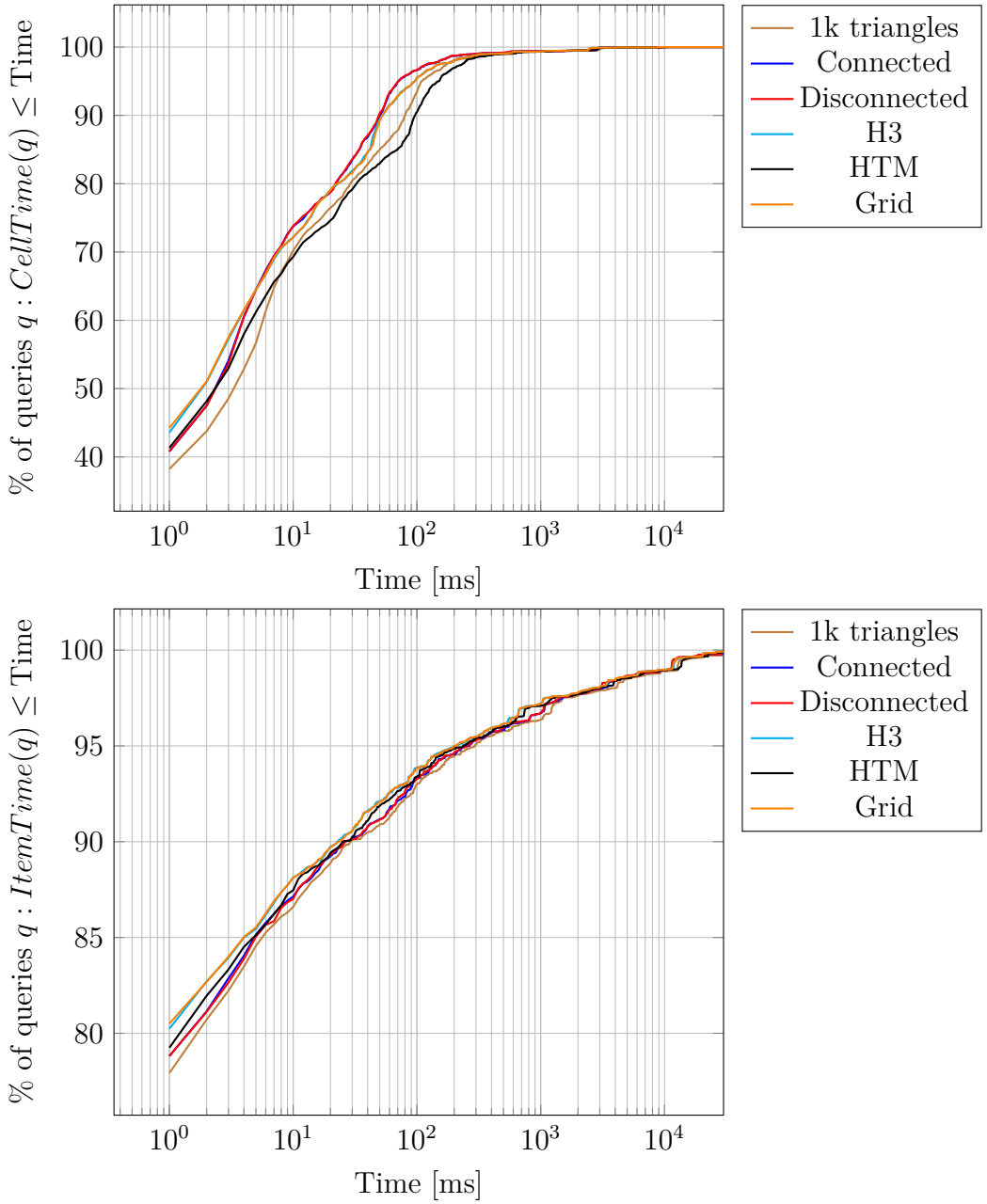


Figure 5.13: Cumulative distribution function of the time it takes to compute the result of the textual website queries containing at least one set operation using various cell arrangements.

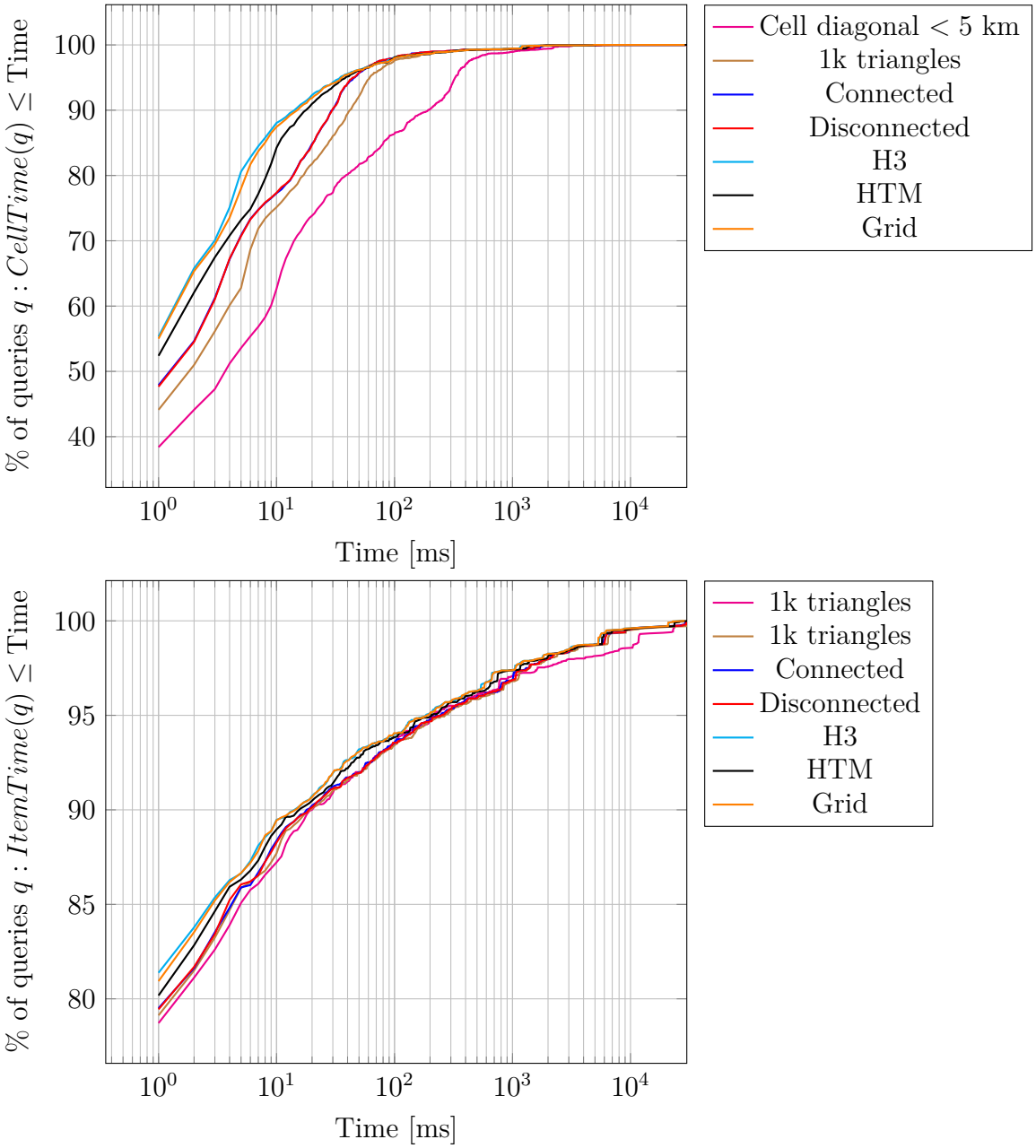


Figure 5.14: Cumulative distribution function of the time it takes to compute the result of the textual website queries containing at least one set operation using various cell arrangements based on the Europe data set.

OSCAR refinement	OSCAR	HTM	H3	Grid
Planet				
All	1923	131	898	773
Disconnected	1842	140	936	807
Connected	1847	145	925	808
Maximal 1 k triangles	1134	201	1270	1120
Europe				
All	979	204	1515	465
Disconnected	943	215	1530	475
Connected	944	222	1522	475
max 1000 triangles	726	299	1618	520
cell diagonal < 5 km	431	441	1731	560
Germany				
All	1064	306	558	654
Disconnected	1047	316	558	661
Connected	1044	316	560	662
max 1000 triangles	804	460	574	744
cell diagonal < 5 km	606	587	590	799
Baden-Württemberg				
All	1038	192	394	230
Disconnected	1014	202	403	235
Connected	1022	198	399	235
max 1000 triangles	923	271	411	249
cell diagonal < 5 km	805	349	436	264

Table 5.18: Number of queries for which the given cell arrangement is the fastest. The column labeled OSCAR show the possibly combined number of queries where a triangulation based arrangement was faster. The rows give the cell refinement types considered for comparison.

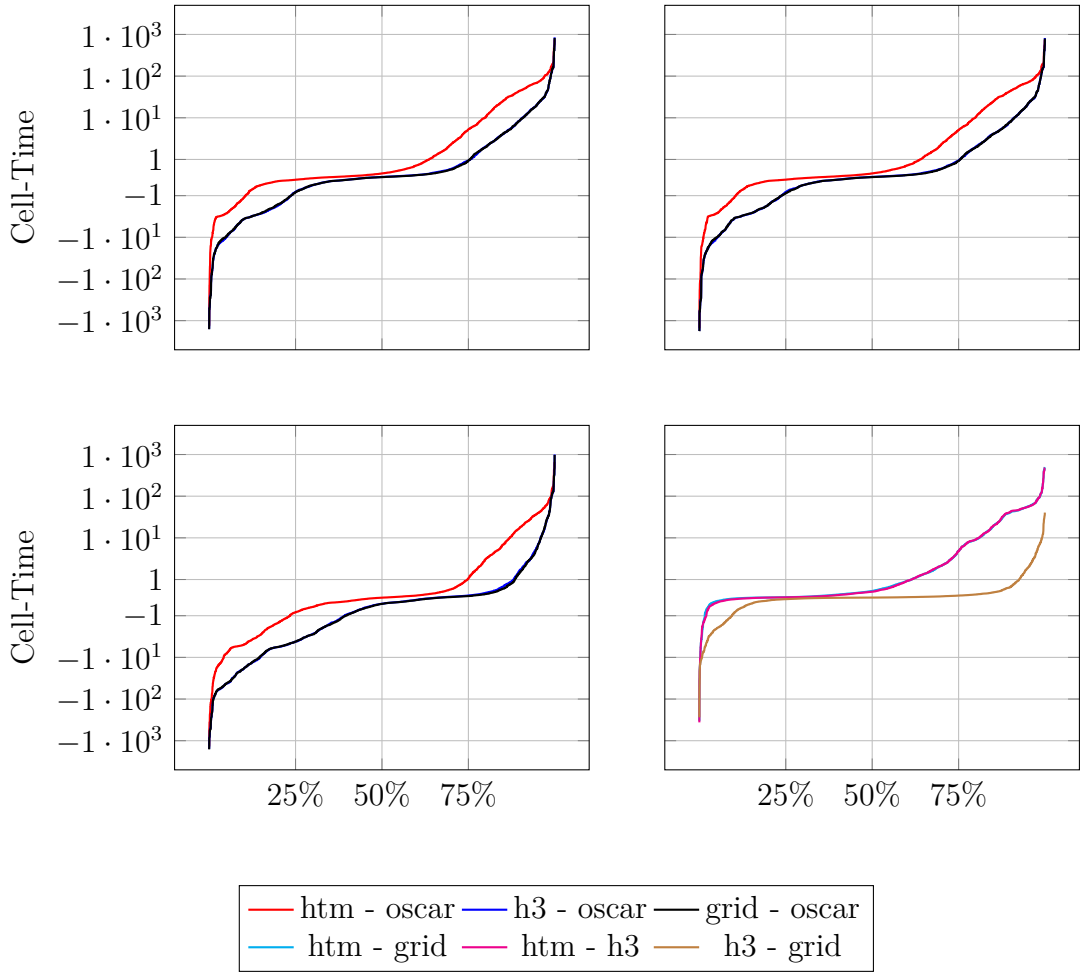


Figure 5.15: Cumulative distribution function of the difference between completion times of the website text queries containing set operations for various cell arrangements based on (refined) triangulations and Discrete Global Grids. Top left: DGG compared to OSCAR with disconnected cells. Top right: DGG compared to OSCAR with connected cells. Bottom left: DGG compared to OSCAR with up to 1000 triangles per cell. Bottom right: DGG compared among themselves. Note that the completion timings of the h3 index and the simple grid are so close together that they overlap for large parts. See section 5.8.1 for an in-depth discussion.

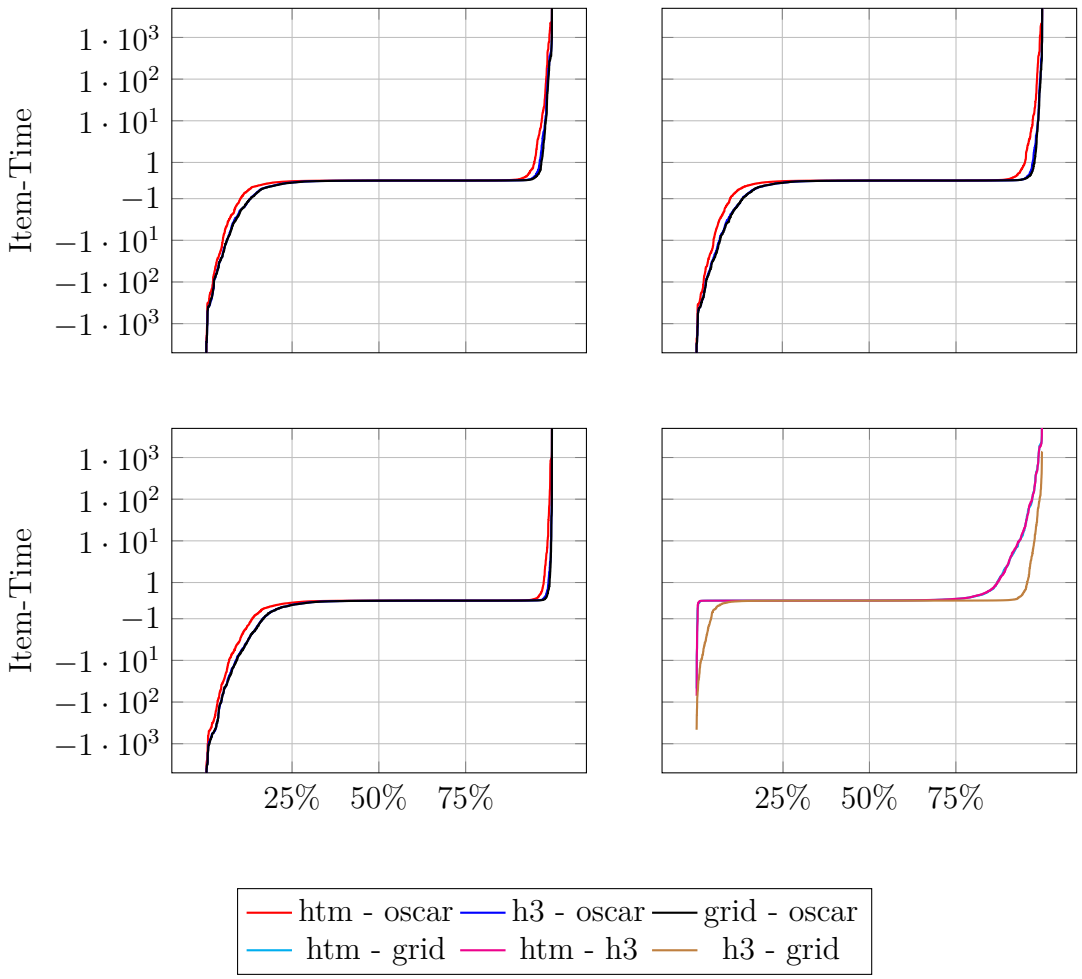


Figure 5.16: Cumulative distribution function of the difference between completion times of the website text queries containing set operations for various cell arrangements based on (refined) triangulations and Discrete Global Grids. Top left: DGG compared to OSCAR with disconnected cells. Top right: DGG compared to OSCAR with connected cells. Bottom left: DGG compared to OSCAR with up to 1000 triangles per cell. Bottom right: DGG compared among themselves. Note that the completion timings of the h3 index and the simple grid are so close together that they overlap for large parts. See section 5.8.1 for an in-depth discussion.

5.9 Comparison to Text-only Search Engines

There exist numerous full-featured frameworks among them ATIRE [106], Galago [71], Lucene [1] and MG4J [24]. Most systems employ compression techniques like variable byte encoding after delta-compression - MG4J being one of the few using a quasi-succinct index to store posting lists [109]. Seven open-source text retrieval framworks were compared in [75] using various text collections where MG4J used the least amount of storage while having competitive query times. In the following we compare the text search capabilities of OSCAR with these two frameworks.

5.9.1 Data Mapping

Lucene as well as MG4J both have the notion of a collection of documents in which one can search. A document is split into fields containing the textual data. Usually a document has at least two fields: a title and a content field. The content of a field is split into tokens, for example into single words or at word boundaries. At query-time one can select the fields which should match a given query. One way to map OpenStreetMap data to documents is to define every item and region as a document and its key-value pairs as fields. For every key there would be one field. Unfortunately this would create a lot of fields. Furthermore this complicates the prefix-search on keys and would likely result in different results compared to OSCAR. Hence we use the following scheme: We have a total of 6 fields per document (c.f. table 5.19). The tokens of a field are then the important values and tags of items and regions together with information inherited from enclosing regions. Additionally we have two fields that are the union of item and region information. This scheme allows us to support almost all text queries available in OSCAR.

Field	Content
1	item values for substring search
2	item tags for prefix search
3	values inherited from enclosing regions
4	tags inherited from enclosing regions
5	union of 1 and 3
6	union of 2 and 4

Table 5.19: Fields of a document for the comparison to Lucene and MG4J.

5.9.2 Implementation

Both Lucene and MG4J are implemented in Java and are available as Java libraries. We used Lucene 6.2.0 and MG4J 5.4.3. OSCAR on the other hand is implemented in C++. To allow for an easy comparison we implemented a small wrapper library using the Java Native Interface to access OSCAR's data structures directly from a Java program. A multi-threaded implementation of the indexing process of Lucene was quite easy since only the call to `addDocument` had to be parallelized. The solution for MG4J on the other hand is more involved. We first create a subindex per thread which indexes a subset of the documents. These subindexes are then merged in a final phase. Merging indexes is part of the normal index creation of MG4J and is therefore quite efficient. Both Lucene and MG4J support advanced text queries and ranking functions which are not available in OSCAR. We therefore removed information like word frequencies, counts and positions keeping only the inverted index and the document pointers. We furthermore disabled OSCAR's support for diacritic in-sensitive search.

5.9.3 Queries

We used the text-only queries from our web-based demonstrator to benchmark Lucene and MG4J. Note that MG4J does not support substring search whereas Lucene does. The storage space of OSCAR could further be reduced if we dropped that capability as well. We removed all spatial queries since Lucene and MG4J do not support these. Additionally MG4J does not support the difference set operations if the index is created without word positions.

5.9.4 Pre-processing

Table 5.20 exhibits the resource usage of OSCAR, Lucene and MG4J to preprocess the data sets. Both Lucene and MG4J need more time compared to OSCAR but are able to index the data set with less memory. Note that MG4J is able to index the data set using less memory in more time as well. Reducing the memory footprint of OSCAR is also possible at the cost of a higher preprocessing time (c.f. 5.3.2) MG4J and Lucene have a rather high CPU utilization indicating a low dependence on disk IO whereas OSCAR is mostly bound by disk IO resulting in a rather low CPU utilization. The final result of both Lucene and MG4J is larger than OSCAR's structures. This would even be true if we only indexed fields 5 and 6.

	Ba-Wü	Germany	Europe	Planet
		OSCAR Global Ids		
Time [h:m]	0:11	1:02	8:22	14:28
Memory [GiB]	35.7	69.8	98.7	146.3
CPU utilization [1]	2.8	3.9	2.8	3.3
Storage size [GiB]	0.8	6.2	38.2	71.4
		OSCAR Local Ids		
Time [h:m]	0:11	0:59	7:26	13:11
Memory [GiB]	36.7	72.4	106.2	159.2
CPU utilization [1]	3.6	3.9	3.0	3.3
Storage size [GiB]	0.6	4.1	26.0	52.3
		Lucene		
Time [h:m]	0:04	12:22	57:41	94:27
Memory [GiB]	53.2	57.2	92.0	123.1
CPU utilization [1]	6.8	43.5	44.0	42.4
Storage size [GiB]	1.1	25.2	367.6	733.1
		MG4J		
Time [h:m]	0:41	3:31	22:05	45:14
Memory [GiB]	50.3	78.2	370.7	416.2
CPU utilization [1]	29.8	30.6	29.5	30.7
Storage size [GiB]	0.9	23.2	312.4	627.0

Table 5.20: Resource usage during construction and final result size for OSCAR, Lucene and MG4J. OSCAR is heavily bound by disk IO as indicated by the low overall CPU-utilization.

	Ba-Wü	Germany	Europe
	OSCAR Global Ids		
Prefix search [s]	23	294	2654
Substring search [s]	41	500	3647
	OSCAR Local Ids		
Prefix search [s]	34	360	3864
Substring search [s]	53	579	2899
	Lucene		
Prefix search [s]	487	10524	-
Substring search [s]	1133	19650	-
	MG4J		
Prefix search [s]	25310	-	-
Substring search [s]	-	-	-

Table 5.21: Time to compute all items of all queries of the textual website queries using a single thread. Note that MG4J does not support substring search. The query times of the Germany data set already exhibits query times suggesting that the Europe and Planet benchmarks would take weeks to compute. For MG4J this is already the case for the Germany data set.

5.9.5 Query processing

Table 5.21 exhibits the timing statistics of the textual queries. Although OSCAR needs less space than Lucene and MG4J it is way faster than these two solutions on the OpenStreetMap data set. The reason for this is in part the efficient handling of regional queries. Consider for example the query "*@amenity:restaurant Stuttgart Germany*". Both Lucene and MG4J produce very large intermediate results that are then intersected whereas OSCAR mostly stays on the cell level which reduces the processing time. Compare this with the cdfs shown in figure 5.17 where we can see that both Lucene and MG4J are kind of competitive with OSCAR for about 90 % of the queries.

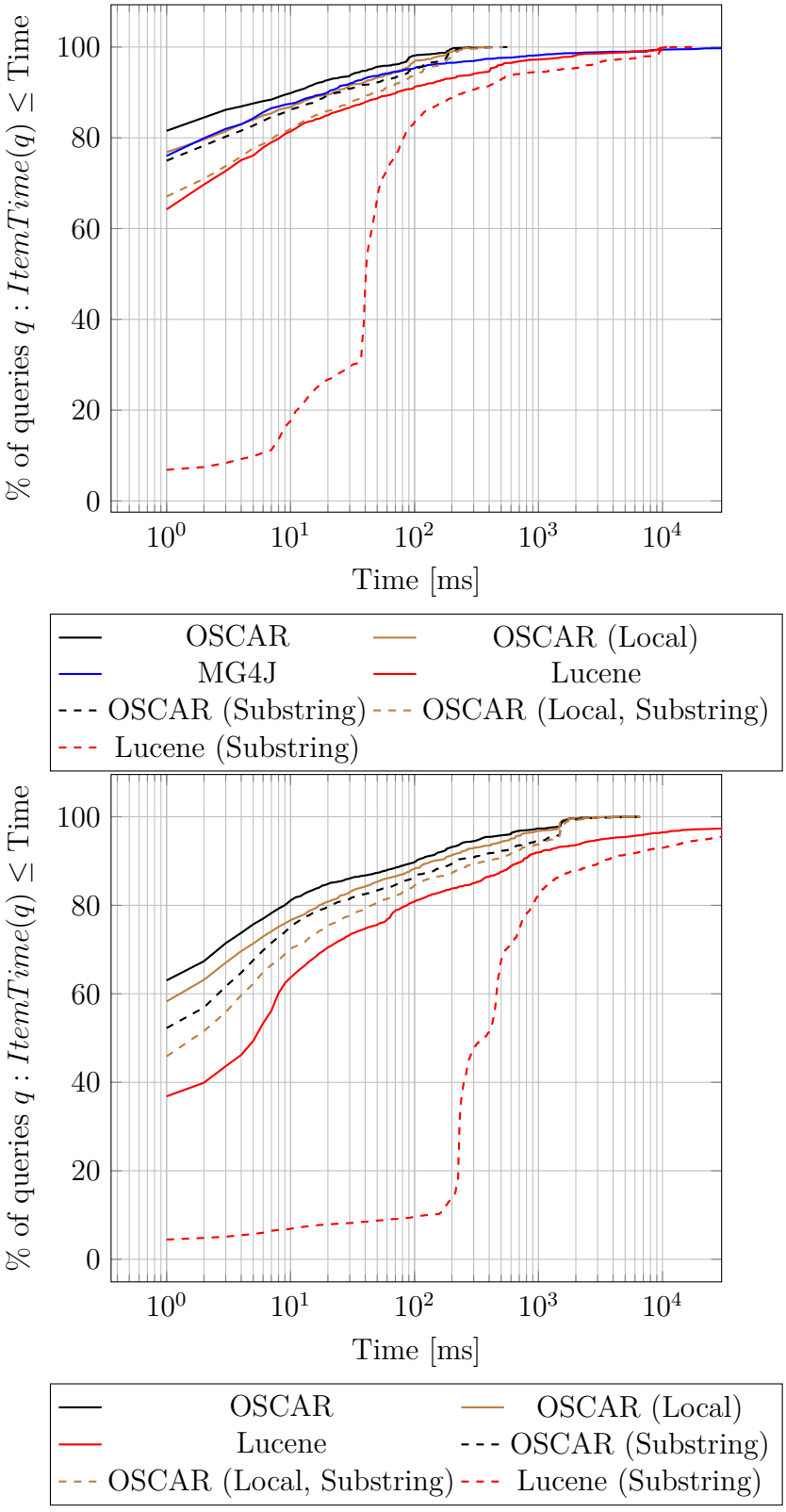


Figure 5.17: Cumulative distribution function of the completion times of the website queries for OSCAR and Lucene. Top: Baden-Württemberg data set. Bottom: Germany data set.

6 Digression: Rational Points on the Unit Sphere

Currently OSCAR uses a very simple triangulation to represent the cell arrangement. We already mentioned some problems that arise with this representation in sections 4.3.2 and 4.3.2. As possible solution we proposed to compute the triangulation on the unit sphere for which we will lay the foundation in the following sections. This chapter is a reproduction of our publication [16] with an improved analysis compared to [17]. We first sketched our algorithm in [13] in order to compute a dynamic Delaunay triangulation on the unit sphere.

Figure 6.1: Spherical Delaunay triangulation (gray) constrained to contain all line segments (black) of streets in Ecuador and the intersection points of constraints (red).

6.1 Introduction

Many mathematical sciences use trigonometric functions in symbolic coordinate transformations to simplify fundamental equations of physics or mathematical systems. However, rational numbers are dominating in computer processing as they allow for simple storage as well as fast exact and inexact arithmetics (e.g. GMP[51], IEEE Float, MPFR[44]). Therefore problems on spherical surfaces often require to scale a point vector, as in choosing a point uniform at random[79], or to evaluate a trigonometric function for a rational angle argument, as in dealing with Geo-referenced data.

A classical theoretical barrier is Niven's theorem[85], which states that the sole rational values of sine for rational multiples of π are $0, \pm 1/2$ and ± 1. The well known Chebyshev polynomials have roots at these values, hence give rise to representations for these algebraic numbers. However, arithmetics in a full algebraic number field might well be too demanding for many applications. For products of sine and cosine, working with Euler's formula on the complex unit circle and Chebyshev polynomials would suffice though.

This manifests in problems of *exact* geometrical computations, since standard methodology relies on Cartesian input[72]. Spheres and ellipsoids are common geometric objects and rational solutions to their defining quadratic polynomials are closely related to Diophantine equations of degree 2. The famous Pythagorean Triples are known to identify the rational points on the circle $\mathbb{S}^1$. Moreover, the unit sphere has a dense set of rational points and so do ellipsoids with rational half-axes through scaling. Spherical coordinates are convenient to reference such Cartesians with angle coordinates and *georeferenced data* denotes points with rational angles. Standard approximations of Cartesians do not necessarily fulfill these equations, therefore subsequent algorithmic results can suffer greatly.

This paper focuses on finding rational points *exactly on* the unit sphere $\mathbb{S}^{d-1} = \left\{x \in \mathbb{R}^d \ : \ \sum_i x_i^2 = 1\right\}$ with bounded distance to the point $x/\|x\|_2$ – its closest point on $\mathbb{S}^{d-1}$. In this work, $x \in \mathbb{R}^d$ can be given by any finite means that allow to compute a rational approximation to it with arbitrary target precision. Using rational Cartesian approximations for spherical coordinates, as derived from MPFR, is just one example of such a black-box model. Moreover, we are interested in calculating rational points on $\mathbb{S}^d$ with small denominators.

6.1.1 Lower Bounds and Instances for Geo-referenced Data on $\mathbb{S}^d$

It is well known in Diophantine approximation that rational numbers have algebraic degree 1 and are hard (in the following qualitative sense) to approximate with other rational numbers. The following folklore observation is an analog to Liouville's lower bound.

Observation 1. *For rational numbers $\frac{a}{b} \neq \frac{p}{q}$, we have*

$$\left|\frac{a}{b} - \frac{p}{q}\right| = \left|\frac{aq - bp}{bq}\right| \geq \frac{1}{bq}$$

If $q < b$, we have a lower bound of $1/q^2$ for rational approximations to $\frac{a}{b}$ with denominators up to q. Pythagorean triples $(x, y, z) \in \mathbb{N}^3$ provide such rational points on $\mathbb{S}^1$, since $(x/z)^2 + (y/z)^2 = 1$. We have a lower bound of $1/z^2$ for approximations with denominators $q < z$. See Section 6.2.4 for rational points on $\mathbb{S}^d$ with the same denominator property.

The situation might look different when dealing with Geo-referenced data (rational angle arguments) only. However, using Chebyshev's polynomials in conjunction with Liouville's lower bound (c.f. Theorem 1) allows to derive explicit constants for Diophantine approximations of $\cos(108°)$.

Given spherical coordinates, the first coordinate of a point on $\mathbb{S}^d$ might well have algebraic values of $r_i = \cos(\frac{i}{5}\pi)$ for $i \in \{1, 2, 3, 4\}$.

$$\begin{aligned}(r_1, r_2, r_3, r_4) &= \left(\frac{1+\sqrt{5}}{4}, \frac{-1+\sqrt{5}}{4}, \frac{1-\sqrt{5}}{4}, \frac{-1-\sqrt{5}}{4}\right) \\ &\approx (+0.8090, +0.3090, -0.3090, -0.8090)\end{aligned}$$

Over $\mathbb{Z}[X]$, the polynomial $U_4(x) = 16x^4 - 12x^2 + 1$ has the irreducible factors

$$U_4(x) = \underbrace{(4x^2 - 2x + 1)}_{=:f(x)}(4x^2 + 2x - 1)$$

Since r_1 and r_3 are the roots of the polynomial f, they have algebraic degree $n = 2$.

Using Liouville's lower bound for r_3, we have for all $\frac{p}{q} \in \mathbb{Q}$

$$\left|r_3 - \frac{p}{q}\right| \geq \frac{\min\{c_2, \frac{1}{c_1}\}}{q^n} \quad ,$$

with constants c_1 and c_2 according to the proof of Liouville's Theorem[76]. The constants $c_1, c_2 > 0$ exist, since the polynomial division of f with the linear factor $(x - r_3)$ results in the continuous function $g(x) = (x - r_1)$. For $c_2 = 1/2 < \sqrt{5}/2$, the interval $I := [r_3 - c_2, r_3 + c_2] \subseteq \mathbb{R}$ is sufficiently small to exclude different roots of f and the inequality

$$\max_{x \in I} \left|g(x)\right| = \max_{x \in I} \left|x - r_1\right| < c_1$$

is met with a generous choice of $c_1 = 2$. This leads to an explicit lower bound on the approximation error to r_3 with denominators q of

$$\left|\cos(108°) - \frac{p}{q}\right| \geq \frac{1}{2 \cdot q^2} \ .$$

6.2 Results

Apart from integers, contemporary computing hardware heavily relies on *floating point numbers*. These are triplets (s, m, e) with $s \in \{0, 1\}$, $m \in \{0, \ldots, 2^l - 1\}$ and

$e \in \{-2^{k-1}+1, \ldots, 2^{k-1}-1\}$. The IEEE standard for Float is $(l,k) = (23,8)$ and $(52,11)$ for Double. The rational number described by such a triplet is

$$\mathrm{val}(s,m,e) = (-1)^s \cdot \begin{cases} \dfrac{2^l+m}{2^l} 2^e & e > 0 \\ \dfrac{2^l+m}{2^l} \dfrac{1}{2^{|e|}} & e < 0 \\ \dfrac{0+m}{2^l} \dfrac{1}{2^{2^{k-1}-2}} & e = 0 \end{cases}$$

where the latter case describes 'denormalized' numbers. In each case, the uncanceled rational value has some power of 2 as the denominator. Since powers of two are the sole divisors of a 2^i, the denominator of the canceled rational has to be a power of two, too. Hence, rational values representable by floating point numbers are a subset of the following set P and fixed-point binary numbers are a subset of P_i:

$$\begin{aligned} \operatorname{img} \mathrm{val} \subseteq \left\{ \frac{z}{2^i} : i \in \mathbb{N}, z \in \mathbb{Z}, z \text{ odd} \right\} &= P \\ \left\{ \frac{z}{2^i} : z \in \mathbb{Z} \right\} = P_i &\subseteq P \quad . \end{aligned}$$

6.2.1 Floating Point Numbers are Insufficient

Fix-point and floating-point arithmetics of modern CPUs work within a subset of rational numbers, in which the denominator is some power of two and the result of each arithmetic operation is 'rounded'.

Theorem 3. *There are only* 4 *floating point numbers on* $\mathbb{S}^1$ *and* 6 *on* $\mathbb{S}^2$.

Proof. We show $\mathbb{S}^{d-1} \cap P^d \nsubseteq \{-1,0,1\}^d$ implies $d \geq 4$. Suppose there is a non-trivial $p \in \mathbb{S}^{d-1} \cap P^d$ with d minimal. Let $x_i/2^{e_i}$ denote the canceled fraction of its i-th coordinate. We have that all $x_i \neq 0$, x_i are odd numbers and all $e_i > 0$ (since p is not one of the $2d$ poles and d is minimal).

W.l.o.g. $e_1 \leq e_2 \leq \ldots \leq e_d$. We rewrite the sphere equation $1 = \sum_{j=1}^{d} (x_i/2^{e_i})^2$ to

$$x_1^2 = 4^{e_1} - \sum_{j=2}^{d} 4^{e_1 - e_j} x_j^2 \quad .$$

For an odd integer y, we have $y^2 = (2k+1)^2 = 4(k^2+k)+1$, leading to the congruence

$$1 \equiv 0 - \sum_{j=2}^{d} \chi_{e_1}(e_j) \qquad \mod 4 \; .$$

Where the characteristic function $\chi_{e_1}(e_j)$ is 1 for $e_1 = e_j$ and 0 otherwise. For $d \in \{2, 3\}$ the right hand side can only have values of $0, -1$ or -2, a contradiction. $\square$

Note that theorem 3 translates to spheres with other radii through scaling. Suppose a sphere in $\mathbb{R}^3$ of radius 2^j has a non-trivial solution $y \in P^3$, then $y/2^j \in P^3$ and would be on $\mathbb{S}^2$, too.

6.2.2 Snapping to Rational Points

We now describe how to compute a good rational approximation *exactly on* the unit sphere $\mathbb{S}^{d-1}$. The input point $x \in \mathbb{R}^d$ can be given by any finite means that allows to compute rational approximations of arbitrary target precision – e.g. rational approximations of Cartesians for spherical coordinates. For the input x, we denote its closest point on $\mathbb{S}^{d-1}$ with $x/\|x\|_2$. The stereographic projection τ and its inverse mapping σ provide $\sigma(\tau(x/\|x\|_2)) = x/\|x\|_2$, since the argument is on $\mathbb{S}^{d-1}$. Instead of determining the value of τ exactly, we calculate an approximation $y \in \mathbb{Q}^d$ and finally evaluate $\sigma(y)$ under exact, rational arithmetics. Hence, the result $\sigma(y)$ is exactly on $\mathbb{S}^{d-1}$. See figure 6.2 for an illustration.

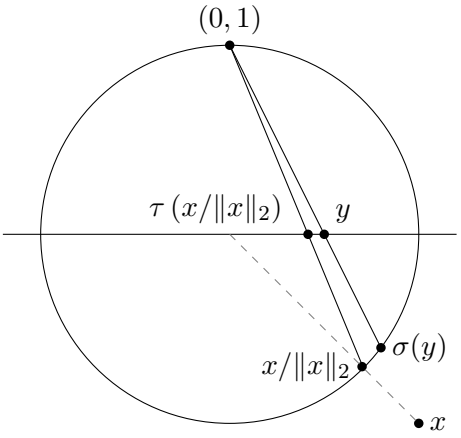


Figure 6.2: Stereographic projection in two dimensions for a point $x/\|x\|_2$ and its rational approximation $\sigma(y)$

The stereographic projection does not preserve distances, leaving it open to bound the approximation error and the size of the resulting denominators. We use the *rotation symmetry* of the sphere to limit the stretching of σ (c.f. Lemma 4): For a non-zero point $x \in \mathbb{R}^d$ we can assume that $i = d$ maximizes $|x_i|$ and $x_d < 0$, otherwise we change the standard orthonormal basis by swapping dimension i and d and using a negative sign for dimension d. Note that such rotations do not

change the actual coordinate values. To keep the *size of denominators* in $\sigma(y)$ small, we use fixed-point arithmetics to determine $y \in \mathbb{Q}^{d-1}$ (c.f. Lemma 5).

Algorithm 1 PointToSphere

In: $x \in \mathbb{R}^d, \quad \varepsilon \in \left(0, \frac{1}{8}\right]$

1. Assert $x_d = \min_i -|x_i|$
2. Choose $y \in \mathbb{Q}^{d-1}$ with $|y_i - \tau_i\left(x/\|x\|_2\right)| \leq \frac{\varepsilon}{2\sqrt{d-1}}$
3. Return $\sigma(y) \in \mathbb{Q}^d$.

See Algorithm 1 for a precise description. Note that the rational point y in statement 2 solely needs to meet the target approximation in the individual coordinates for

$$\tau_i(x/\|x\|_2) = \frac{x_i}{\|x\|_2 - x_d} \quad .$$

Generally, this can be determined with methods of 'approximate expression evaluation' to our target precision[72]. If x is an approximation to a georeferenced point, this denominator is well conditioned for calculations with multi-precision floating-point arithmetics[28, 44]. Using exact rational arithmetics for statement 3, we obtain a rational Cartesian coordinates *on* the unit sphere.

Observation 2. *For $d > 1$ and $x \in \mathbb{S}^{d-1}$ with $x_d = \min_i -|x_i|$, we have*

$$\|\tau(x)\|_2 \leq \sqrt{\frac{\sqrt{d}-1}{\sqrt{d}+1}} < 1 \quad .$$

Proof. Using $x_d = \min_i -|x_i|$ and $\sum_i x_i^2 = 1$, we have the bounds $1/d \leq x_d^2 \leq 1$ and

$$\|\tau(x)\|_2^2 = \frac{\sum_{i=1}^{d-1} x_i^2}{(1-x_d)^2} = \frac{1-x_d^2}{(1-x_d)^2} = \frac{1+x_d}{1-x_d} \leq \frac{1-1/\sqrt{d}}{1+1/\sqrt{d}} \, .$$

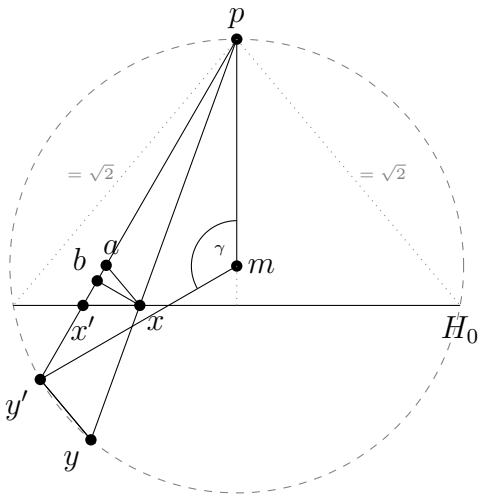
Where the latter term is in $(0, 1)$ for any d. □

Hence the $(d-1)$-ball $\mathbf{B}_1^{d-1} = \{x \in \mathbb{R}^d \ : \ \|x\|_2 \leq 1\}$ contains $\tau(x)$.

6.2.3 Approximation Quality

See [17] for an earlier version of this paper with a weaker, but elementary, analysis.

We consider the problem in the 2D hyperplane $H_{pyy'}$, defined by two points $y = \sigma(x)$, $y' = \sigma(x')$ on $\mathbb{S}^{d-1}$ and the projection pole $p \in \mathbb{R}^d$. Given the rotation step in Algorithm 1, the projection plane $H_0 = \{x \in \mathbb{R}^d : x_d = 0\}$ separates p and y, y' in $\mathbb{R}^d$ and in $H_{pyy'}$. Since each $q \in H_0 \cap S^{d-1}$ has $\|q - p\|_2 = \sqrt{2}$ (consider $\overline{pq}$ in H_{0pq}), the circumcircle C of p, y and y' contains exactly two of these points. Hence, the line of $H_{pyy'} \cap H_0$ is orthogonal to the circumcircle's diameter through p. Moreover, the circles diameter is in $[\sqrt{2}, 2]$. We denote with x the point that is closer to p in $H_{pyy'}$, meaning $\|x\|_2 \leq \|x'\|_2$. Note that x' and x can be on the same or opposite circumcircle halves.



In this section we denote with $B = \overline{bx}$ the perpendicular from x on $\overline{py'}$,$E = \overline{xx'}$, $L = \overline{yy'}$ and $L_x = \overline{xa}$ its triangle scaled version meeting x. Note that B and L_x are above H_0, hence above E.

Lemma 3. *For $x, x' \in \mathbf{B}_1^{d-1}$ with $\|x\|_2 \leq \|x'\|_2$, we have*

$$\frac{\|p - x\|_2}{\|p - \sigma(x)\|_2} \|\sigma(x) - \sigma(x')\|_2 \leq \|x - x'\|_2 \ .$$

Proof. We show $L_x \leq E$ by proofing $\alpha \leq \beta$ for the two angles

$$\beta := \measuredangle bxx'$$
$$\alpha := \measuredangle axb \ .$$

The inner angle sum of $\triangle xab$ with a supplementary angle argument and triangle scaling provide $\measuredangle py'y = 90° + \alpha$. Let m denote the center of C. Since $\overline{pm}$ is

orthogonal on H_0 and $\measuredangle bx'x = 90° - \beta$, we have $\measuredangle mpx' = \beta$. In the isosceles triangle $\triangle py'm$, the central angle $\gamma = 180° - 2\beta$. Fixing arc $\overline{py'}$ on C for the inscribed angle theorem provides $\measuredangle y'yp = \gamma/2$.

Now, suppose $\alpha > \beta$. The inner angle sum of $\triangle pyy'$ states

$$\begin{aligned} 0 \leq \measuredangle y'py &= 180° - \measuredangle y'yp - \measuredangle py'y \\ &= 180° - \measuredangle y'yp - (90° + \alpha) \\ &= 180° - \gamma/2 - (90° + \alpha) \\ &= -\alpha + \beta \end{aligned}$$

a contradiction. □

Lemma 4. *For $x, x' \in \mathbf{B}_1^{d-1}$, we have*

$$\left\| \sigma(x) - \sigma(x') \right\|_2 \leq 2 \, \|x - x'\|_2 .$$

Proof. Using Lemma 3, we have $L_x \leq E$ and the statement follows via triangle scaling:

$$L = L_x \, \overline{py} \, / \, \overline{px} \leq 2L_x \leq 2E \ ,$$

since $\overline{px} \geq 1$ and $\overline{py} \leq 2$. □

This statement is tight, considering the two points $x = 0$ and $x' = \left(\varepsilon/\sqrt{d-1}, \ldots, \varepsilon/\sqrt{d-1}\right)$. We have $\|x - x'\|_2 = \varepsilon$ and $\|\sigma(x) - \sigma(x')\|_2 = 2\frac{1}{\sqrt{1+\varepsilon^2}}\varepsilon$.

Theorem 4. *Algorithm 1 calculates an ε-approximation exactly on the unit sphere.*

Proof. Let $x^* = x/\|x\|_2$ and $\sigma(y)$ denote the result. Given the rotation, x^* holds for Observation 2. Hence, we can use Lemma 4 to derive

$$\begin{aligned} \|\sigma(y) - x^*\|_\infty &= \|\sigma(y) - \sigma(\tau(x^*))\|_\infty \\ &\leq \|\sigma(y) - \sigma(\tau(x^*))\|_2 \\ &\leq 2\|y - \tau(x^*)\|_2 \\ &\leq 2\sqrt{(d-1)\frac{\varepsilon^2}{4(d-1)}} = \varepsilon \end{aligned}$$

as upper bound on the approximation error. □

This analysis is rather tight, as demonstrated by the red curve and points in Figure 6.3.

6.2.4 Denominator Sizes

We now describe a relation between rational images of σ and the lowest common multiple of denominators of its rational preimages. This leads to several strategies for achieving small denominators in the results of Algorithm 1.

Lemma 5 (Size of images under σ)**.** *Let $x \in \mathbb{Q}^{d-1} \cap \mathbf{B}_1^{d-1}$ with $x_i = p_i/q_i$ and $Q = lcm(q_1, \ldots, q_{d-1})$ be the lowest common multiple, then*

$$\sigma_k(x) = \frac{n_k}{m}$$

with integers $n_i, m \in \{-2Q^2, \ldots, 2Q^2\}$ for all $1 \leq k \leq d$.

Proof. Let $q_i' \in \{1, \ldots, Q\}$ such that $q_i' \cdot q_i = Q$ for all i. Since the formula of σ is similar in all but the last dimension, we describe the following two cases. For $k = d$, we have

$$\sigma_k(x) = \frac{-1 + \sum_{i=1}^{d-1} p_i^2/q_i^2}{1 + \sum_{i=1}^{d-1} p_i^2/q_i^2} = \frac{-Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2}{Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2} =: \frac{n_k}{m}$$

Using the bound $x \in \mathbf{B}_1^{d-1}$, we have $0 \leq \sum_{i=1}^{d-1} q_i'^2 p_i^2 \leq Q^2$ and we derive for n_k and m

$$|n_k| = \Big| -Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2 \Big| \leq Q^2$$

$$m = Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2 \leq 2Q^2$$

For $k < d$, we have

$$\begin{aligned}\sigma_k(x) &= \frac{2p_k/q_k}{1 + \sum_{i=1}^{d-1} p_i^2/q_i^2} \\ &= \frac{Q^2 \cdot 2p_k/q_k}{Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2} \\ &= \frac{Qq_k' \cdot 2p_k}{Q^2 + \sum_{i=1}^{d-1} q_i'^2 p_i^2} =: \frac{n_k}{m}\end{aligned}$$

Using the bound $x \in \mathbf{B}_1^{d-1}$, we have that each $|p_i| \leq q_i$ and this bounds $|n_k| = Qq_k' \cdot 2|p_k| \leq 2Q^2$. We already discussed the bound on m in the first case. □

Note that we apply this lemma in practice with fixed-point binary numbers $p_i/q_i \in P_s$. Meaning all $q_i = 2^s = Q$ for some significant size s.

Theorem 5. *Denominators in ε-approximations of Algorithm 1 are at most*

$$\frac{10(d-1)}{\varepsilon^2} .$$

Proof. Using standard multi-precision floating point arithmetics allows to derive rational values y, with denominators that are $Q = \lceil \frac{2\sqrt{d-1}}{\varepsilon} \rceil$. Using $\varepsilon \leq 1/8$ and Lemma 5 bounds the size of the denominators in images σ with

$$\begin{aligned} 2Q^2 &\leq 2\left(1 + \frac{2\sqrt{d-1}}{\varepsilon}\right)^2 \\ &= \frac{2}{\varepsilon^2}\left(\underbrace{\varepsilon^2 + \varepsilon 4\sqrt{d-1}}_{\leq (d-1)} + 4(d-1)\right) . \end{aligned}$$

□

For certain dimensions and in practice(c.f. Section 6.4.1), we can improve on the simple usage of fixed-point binary numbers. For $\mathbb{S}^1$ we can rely on the continued fraction algorithm to derive rational approximations of $\alpha = \tau(x/\|x\|_2)$ with $|\alpha - p/q| \leq 1/2q^2$. Using this in Algorithm 1 leads to approximations with $\varepsilon = 1/q^2$ on the circle $\mathbb{S}^1$ with denominators of at most $2q^2$.

Note that for $\mathbb{S}^d$ with $d \geq 2$ one can rely on algorithms for simultaneous Diophantine approximations (c.f. Theorem 2) to keep the lowest common multiple Q in Lemma 5 small. Note that it might well be *simpler* to find Diophantine approximations with small Q.

There have been many approaches to find generalizations of the continued fraction algorithm for $d > 1$. One of the first approaches is the Jacobi-Perron algorithm, which is rather simple to implement[105](c.f. Section 6.4.1). More advanced approaches [87] rely on the LLL-algorithm for lattice basis reduction[70]. For $d = 2$ there is an algorithm to compute all Dirichlet Approximations[62], which we find hard to oversee given its extensive presentation. Moreover, their experimental comparison shows that the Jacobi-Perron algorithm is practically well suited for $d = 2$.

We close this section with a transfer result of Theorem 2 with our Theorem 4 and Lemma 5.

Corollary 1. *Let $x \in \mathbb{S}^{d-1}$ and $N \in \mathbb{N}$. There is $p \in \mathbb{Z}^{d-1}$ and $q \in \{1, \ldots, N\}$ with*

$$\left\| x - \sigma\left(\frac{1}{q}p\right) \right\|_\infty \leq \frac{2\sqrt{d-1}}{q\sqrt[d-1]{N}}$$

and all denominators of $\sigma\left(\frac{1}{q}p\right)$ are at most $2q^2$.

This existence statement allows for brute-force computations. However, we just use it for comparisons in Section 6.4.1.

6.3 Implementation

Apart from [30] for $\mathbb{S}^2$, most implementations of spherical Delaunay triangulations are not 'stable'. Approaches based on d-dimensional convex hull algorithms produce only a tessellation for input not exactly *on* $\mathbb{S}^{d-1}$. (c.f. Section 1.2.7)

Few available implementations allow dynamic point or constraint insertion and deletion – not even in the planar case of $\mathbb{R}^2$. The 'Computational Geometry Algorithms Library' (CGAL [91]) is, to our knowledge, the sole implementation providing dynamic insertions/deletions of points *and* constraint line segments in $\mathbb{R}^2$.

With [19], we provide open-source implementations of Algorithm 1 for $\mathbb{S}^d$. In [18], we provide an implementation for spherical Delaunay triangulations on $\mathbb{S}^2$ with ε-stable constructions of intersection points of constraint line-segments (c.f. Section 6.3.2).

6.3.1 RATional Sphere Snapping for $\mathbb{S}^d$

Libratss is a C++ library which implements Algorithm 1, based on the open-source GMP library for exact rational arithmetics [51] and the GNU 'Multiple Precision Floating-Point Reliably'(MPFR) library[44]. The implementation allows both, input of Cartesian coordinates of arbitrary dimension and spherical coordinates of $\mathbb{S}^2$. Note that this implementation allows geometric algorithms, as for d-dimensional convex hull, to rely on rational input points that are *exactly* on $\mathbb{S}^{d-1}$. In light of the discussion on the denominator sizes in Section 6.2.4, we provide two additional strategies to fixed-point snapping, as analyzed in Theorem 4. We implemented the Continued Fraction Algorithm to derive rational ε-approximations with small denominators and the Jacobi-Perron algorithm for $\mathbb{S}^2$. The library interface also allows to automatically chose the approximation method which results in smaller denominators, approximation errors or other objectives, like byte-size.

6.3.2 Incremental Constrained Delaunay Triangulation on $\mathbb{S}^2$

Libdts2 implements an adapter for the *dynamic constraint* Delaunay triangulation in the Euclidean plane $\mathbb{R}^2$ of CGAL. Since this implementation requires an initial outer face, we introduce an small triangle, that only contains the north-pole, to allow subsequent insertions of points and constraints. For points *exactly* on the unit sphere, the predicate **'is A in the circumcircle of B, C and D'** reduces to the well studied predicate **'is A above the plane through B, C and D'**. The

implementation overloads all predicate functions accordingly and uses Algorithm 1 for the construction of rational points on the sphere for intersections of Great Circle segments.

ε-stable geometric constructions

Any means of geometric construction that allows to approximate a certain point, can be used as input for Algorithm 1 – e.g. the intersection of Great Circle segments. Consider two intersecting segments of rational points on $\mathbb{S}^2$. The two planes, containing the segments and the origin as a third point, intersect in a straight line. Each (rational) point on this line can be used as input for our method, as they identify the two intersection points on the sphere. Using such input for Algorithm 1 allows simple schemes to derive stable geometric constructions of rational points on $\mathbb{S}^d$ within a distance of ε to the target point.

6.4 Experiments

We used real world and synthetic data for our experiments. Geo-referenced data was sampled from regional extracts from the OpenStreetMap project[2], as of January 26th, 2017. Random Cartesian coordinates of points on $\mathbb{S}^d$ were created with the uniform generator 2 of [79]. All benchmarks were conducted on a single core of an Intel Xeon E5-2650v4. Peak memory usage and time were measured using the `time` utility.

6.4.1 Approximation Quality and Size

We experimentally analyze the actual approximation error in results of Algorithm 1 for several levels of ε using the MPFR library. In this section e denotes the significands required in statement 2 of Algorithm 1 for the required result precision ε. This is

$$e = \left\lceil -\log_2 \left(\frac{\varepsilon}{2\sqrt{d-1}} \right) \right\rceil \quad .$$

We simply setup the MPFR data types with significand sizes up to 1024 bits, and conducted our experiments on much lower levels of e. This allows us to derive some 'measure' of the actual approximation errors of our method.

We analyzed the approximation errors δ and denominator bit-sizes q for 100 random points on $\mathbb{S}^2$. Figure 6.3 compares the results of our algorithm under several levels of target precision e and strategies for statement 2 in our method. The magenta line indicates the quality and size of the approach in [101]. The red line indicates the bounds of our Theorems 4 and 5 on the fixed-point strategy, while the yellow line indicates the bound of Corollary 1. Note that results using

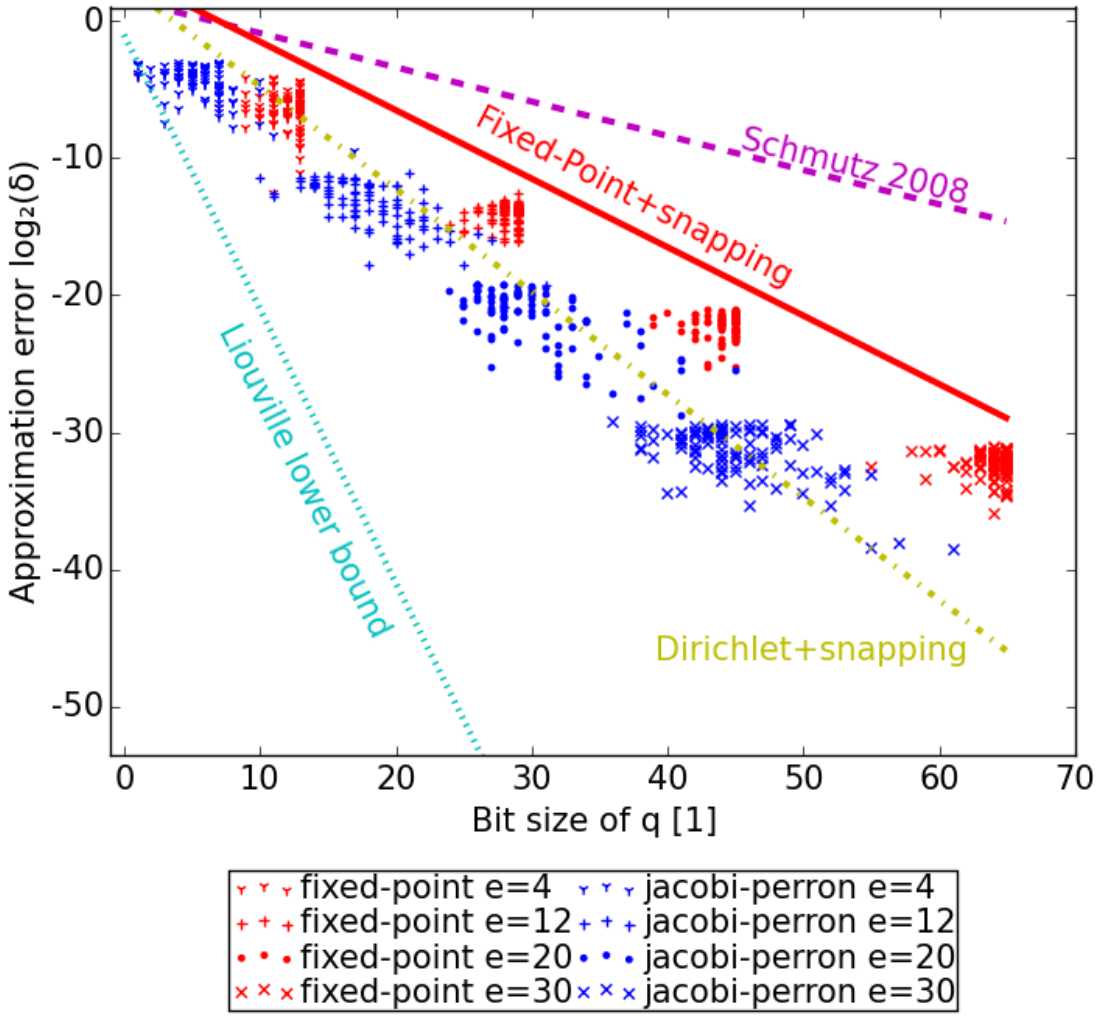


Figure 6.3: Approximation quality and denominator size of 100 random points on $\mathbb{S}^2$ for various levels of target precision e and approximation strategies (red, blue) of Algorithm 1. Theoretic bounds are indicated with lines.

the Jacobi-Perron strategy (blue dots) allows our method to further improve on the fixed-point strategy (red dots). Note that we use Liouville's lower bound as statement on the approximability of a worst-case point. There might well be points of higher algebraic degree that allow better approximations (c.f. Section 6.1.1).

Table 6.1 exhibits average approximation errors δ, denominator bit-sizes q and the computation time t of our method for millions of points. Synthetic data sets have several dimensions, while the real world data sets have dimension 3. For $\mathbb{S}^2$, we provide comparison of the fixed-point strategy (fx) with the Jacobi-Perron strategy (jp) of our method. Using $e = 31$ is sufficient to obtain results *exactly on* $\mathbb{S}^2$ with a δ of less than 1cm, relative to a sphere with radius of the earth. This is enough for most applications dealing with spatial data *and* allows storage within the word size of contemporary computing hardware. This allows practical applications on $\mathbb{S}^2$ to store 4 integer long values for the 3 numerators and the common denominator (c.f. Lemma 5) occupying 32 bytes. Note that storing 3 double values occupies 24 bytes but *cannot* represent Cartesian coordinates *exactly on* the sphere.

6.4.2 Constrained Delaunay Triangulation with Intersection Constructions

A Constrained Delaunay Triangulation of a point set contains required line-segments as edges, but is as close to the Delaunay triangulation as possible [32]. We used *very large* street networks of several regions from the OpenStreetMap project for points and constraint edges – e.g. each line-segment of a street is an edge in the result triangulation. Since $\sim 0.5\%$ of the line-segments in these data sets intersect, we approximated the intersection points using $e = 31$ for Algorithm 1. Table 6.2 exhibits total running time, peak memory usage and the result sizes of our `libdts2` implementation. Small data sets like Saarland and Germany allow quick calculation on a recent workstation computer. See Figure 6.1 for the Ecuador data set. Note that the current implementation has a storage overhead for each point, as we keep the results of the GMP library rather than truncating to integers of architectures word size. Computing the triangulation for the planet data set was only possible on rather powerful hardware with at least 768 Gigabytes of memory taking less than a quarter of a day.

		Germany	Planet	u.a.r $\mathbb{S}^2$	u.a.r $\mathbb{S}^9$	u.a.r $\mathbb{S}^{99}$
dimension		3	3	3	10	100
size $[10^3]$		2,579.6	3,702.4	1,000.0	1,000.0	100.0
e=23						
	$\delta[m]$	0.7	0.7	0.7	1.0	3.2
fx	q [1]	46.0	46.0	46.0	46.0	46.0
	t $[\mu s]$	17	16	16	117	546
	$\delta[m]$	0.4	0.4	0.5	-	-
jp	q [1]	33.6	34.2	34.1	-	-
	t $[\mu s]$	63	57	58	-	-
e=31						
	$\delta[m]$	2.7e-3	2.6e-3	2.8e-3	4.0e-3	12.6e-3
fx	q [1]	62.0	62.0	62.0	62.0	62.0
	t $[\mu s]$	17	16	17	118	554
	$\delta[m]$	1.7e-3	1.7e-3	1.8e-3	-	-
jp	q [1]	45.2	45.8	45.8	-	-
	t $[\mu s]$	77	72	73	-	-
e=53						
	$\delta[m]$	6.3e-10	6.2e-10	6.6e-10	9.6e-10	30.1e-10
fx	q [1]	106.0	106.0	106.0	106.0	106.0
	t $[\mu s]$	16	16	17	118	548
	$\delta[m]$	3.9e-10	3.9e-10	4.3e-10	-	-
jp	q [1]	77.2	77.8	77.7	-	-
	t $[\mu s]$	118	111	112	-	-
e=113						
	$\delta[m]$	5.5e-28	5.4e-28	5.7e-28	8.3e-28	26.1e-28
fx	q [1]	226.0	226.0	226.0	226.0	226.0
	t $[\mu s]$	19	19	19	126	617
	$\delta[m]$	3.4e-28	3.4e-28	3.7e-28	-	-
jp	q [1]	164.5	165.1	165.1	-	-
	t $[\mu s]$	219	218	220	-	-

Table 6.1: Mean-values of approximation error δ $[m]$, denominator bit-size q [1] and computation time t $[\mu s]$ for synthetic and real-world point sets for various dimensions and levels of target precision e. The Jacobi-Perron strategy is denoted by 'jp' and the fixed-point strategy by 'fx'.

	Saarland	Germany	Europe	Planet
Input				
Segments [10^6]	0.32	25.75	222.92	668.61
Output				
Vertexes [10^6]	0.29	24.45	213.01	634.42
Edges [10^6]	0.87	73.37	639.04	1,903.27
Faces [10^6]	0.58	48.91	426.03	1,268.84
Resource usage				
Time [h:m]	< 0:01	0:05	0:49	5:08
Memory [GiB]	< 0.4	27.7	243.2	724.1

Table 6.2: Time and memory usage to compute spherical Delaunay triangulations for OpenStreetMap data sets.

7 Open Problems

In the following we would like to discuss some future extensions and research directions.

Distributed Computing The cell based approach should allow for a distributed computation variant with high efficiency both for the creation of the data structures as well as for the query. For the creation all that is needed is an implementation of constrained Delaunay triangulation for distributed computation. Unfortunately we are not aware of any available implementation. One possible solution would be to partition the data set according to the countries of the world. The problem with this approach is that there are a lot of regions like "The Alps" intersecting multiple countries.

Updates Another important aspect are dynamic updates. Our current data structure implementation uses many compression techniques to reduce the storage footprint. This however makes it extremely difficult to update the data without rewriting large parts. Changing the storage layout to allow for fast incremental updates is therefore mandatory. Since this would increase the storage footprint a distributed variant would be advisable — which was not the primary goal of our implementation.

Hierarchical Cell Arrangement A hierarchical approach may further improve processing speeds for some queries. Consider for example the query "*#"Germany" @highway @amenity @building*". The delayed processing scheme already reduces the processing time by removing all cells outside of Germany. However if we take a look at the Inclusion-DAG then we see that Germany is just a single node covering thousands of cells. A simple variant would be to use a subset of the Inclusion-DAG that is a tree and do the set operation based on this tree. Each query statement would then return a subset of that tree which allows us to compute the set operations in a recursive manner. In the case of the aforementioned intersection the tree of the "*#"Germany*"" query would prune most nodes of the other query statements high up in the hierarchy. However there certainly are queries that would benefit from other subsets of the Inclusion-DAG. A tree independent of the Inclusion-DAG is another option, for example based on the spatial grids introduced in 2.3.1.

Cell Arrangement Complexity So far we only control for the size or number of triangles of a cell. This however may unnecessarily add more vertexes to the triangulation. It may instead be of interest to reduce the number of vertexes which also reduces the number of triangles. To accomplish this task we need to simplify the borders of the cells while keeping the topology of the cell arrangement. One important additional constraint is that the items them-selves are part of the topology and hence we need to compute a topology preserving simplification of millions of polygonal line segments while obeying billions of additional topology constraints introduced in the form of points by the items. One possible solution may be the algorithm introduced in [81] which provides a solution for our problem hopefully in a reasonable amount of time.

Alternative Textual Spatial Indexes OSCAR is designed to support different kinds of indexing structures. Unfortunately most of the alternative textual spatial indexes introduced in section 1.2.3 do not publish any kind of source code. We are therefore interested in implementing the basic building blocks like R-trees, quadtrees, inverted index etc. using generic programming. Based on this library it should be fairly easy to implement the alternative indexing schemes. Additionally a standardized set of benchmarks should be part of the library thus enabling other researchers to compare their new structures with existing ones using the same benchmark methodology.

Spherical Geometry in OSCAR From a practical point of view, it is of great interest to bound the storage size of denominators to a maximum of 64 bits – the word size of current computing architectures. Preliminary research suggests the viability of this approach with a reduction from 545 GiB down to 109 GiB for the planet test data set listed in table 6.2. We can take this a step further by storing points on the projection plane using only three 32 bit numbers instead of four 64 bit numbers. Predicates involving points in the same projection plane can then be calculated in the two-dimensional space of the projection plane. These points could easily be stored in our static out-of-memory data structures while increasing the storage size only slightly. Further reductions in storage space or increases in approximation accuracy may be possible by using advanced algorithms for simultaneous approximation, like the LLL-algorithm or the Dirichlet approximation algorithm for $\mathbb{S}^2$.

For the theoretical part, we are interested if finding simultaneous rational approximations with small lowest common multiple of the denominators is simpler than finding Dirichlet approximations. We are also interested in generalizing the method to provide rational approximations with small *absolute* errors on ellipsoids with rational semi-principal axes – e.g. the geographic WGS84 ellipsoid.

8 Appendix

8.1 Sample Queries used in the Introduction

```
/*
This has been generated by the overpass-turbo wizard.
The original search was:“
waterway=waterfall in "Germany
*/
[out:json][timeout:25];
// fetch area “"Germany to search in
{{geocodeArea:Germany}}->.searchArea;
// gather results
(
// query part for: “waterway="waterfall
node["waterway"="waterfall"](area.searchArea);
way["waterway"="waterfall"](area.searchArea);
relation["waterway"="waterfall"](area.searchArea);
);
// print results
out body;
>;
out skel qt;
```

Code Listing 8.1: The query used produce the result for overpass-turbo depicted in figure 1.2

8.2 Query Language

The following grammar shows the language supported by OSCAR.

⟨*Query*⟩ ::= ⟨*Unary-Op*⟩ ' ' ⟨*Query*⟩
  | ⟨*Query*⟩ ' ' ⟨*Binary-Op*⟩ ' ' ⟨*Query*⟩
  | '(' ⟨*Query*⟩ ')'
  | ⟨*Spatial-Query*⟩
  | ⟨*Text-Query*⟩

⟨*Unary-Op*⟩ ::= ⟨*Full-Match-Conversion*⟩
  | ⟨*Dilation*⟩
  | ⟨*Compass*⟩

⟨*Binary-Op*⟩ ::= ⟨*Between*⟩ | ⟨*Set-Op*⟩

⟨*Full-Match-Conversion*⟩ ::= '%'

⟨*Dilation*⟩ ::= '%' ⟨*Number*⟩ '%'

⟨*Compass*⟩ ::= ':^' | ':north-of'
  | ':>' | ':east-of'
  | ':v' | ':south-of'
  | ':<' | ':west-of'

⟨*Between*⟩ ::= '<->'

⟨*Set-Op*⟩ ::= ⟨*Intersection*⟩
  | ⟨*Union*⟩
  | ⟨*Difference*⟩

⟨*Intersection*⟩ ::= ' ' | '/'

⟨*Union*⟩ := '+'

⟨*Difference*⟩ ::= '-'

⟨*Spatial-Query*⟩ ::= ⟨*Rectangle*⟩
  | ⟨*Polygon*⟩
  | ⟨*Path*⟩
  | ⟨*Point*⟩

⟨*Text-Query*⟩ ::= ⟨*Region+Item-Query*⟩
| ⟨*Region-Query*⟩
| ⟨*Item-Query*⟩

⟨*Rectangle*⟩ ::= '`$rect:`' ⟨*Coordinate*⟩ '`,`' ⟨*Coordinate*⟩

⟨*Polygon*⟩ ::= '`$poly:`' ⟨*Coordinate*⟩ ['`,`' ⟨*Coordinate*⟩]

⟨*Path*⟩ ::= '`$path:`' ⟨*Coordinate*⟩ '`,`' ⟨*Coordinate*⟩]

⟨*Point*⟩ ::= '`$point:`' ⟨*Coordinate*⟩

⟨*Region+Item-Query*⟩ ::= ⟨*String-Query*⟩

⟨*Region-Query*⟩ ::= '`#`' ⟨*String-Query*⟩

⟨*Item-Query*⟩ ::= '`!`' ⟨*String-Query*⟩

⟨*String-Query*⟩ ::= '`"`' ⟨*String*⟩ '`"`'
| ⟨*String*⟩ '`?`'
| '`?`' ⟨*String*⟩ '`?`'
| '`?`' ⟨*String*⟩
| ⟨*String*⟩

⟨*Coordinate*⟩ ::= ⟨*Latitude*⟩ '`,`' ⟨*Longitude*⟩

8.3 Experiments

The following pages show the query timings for all data sets not listed in the experiments chapter.

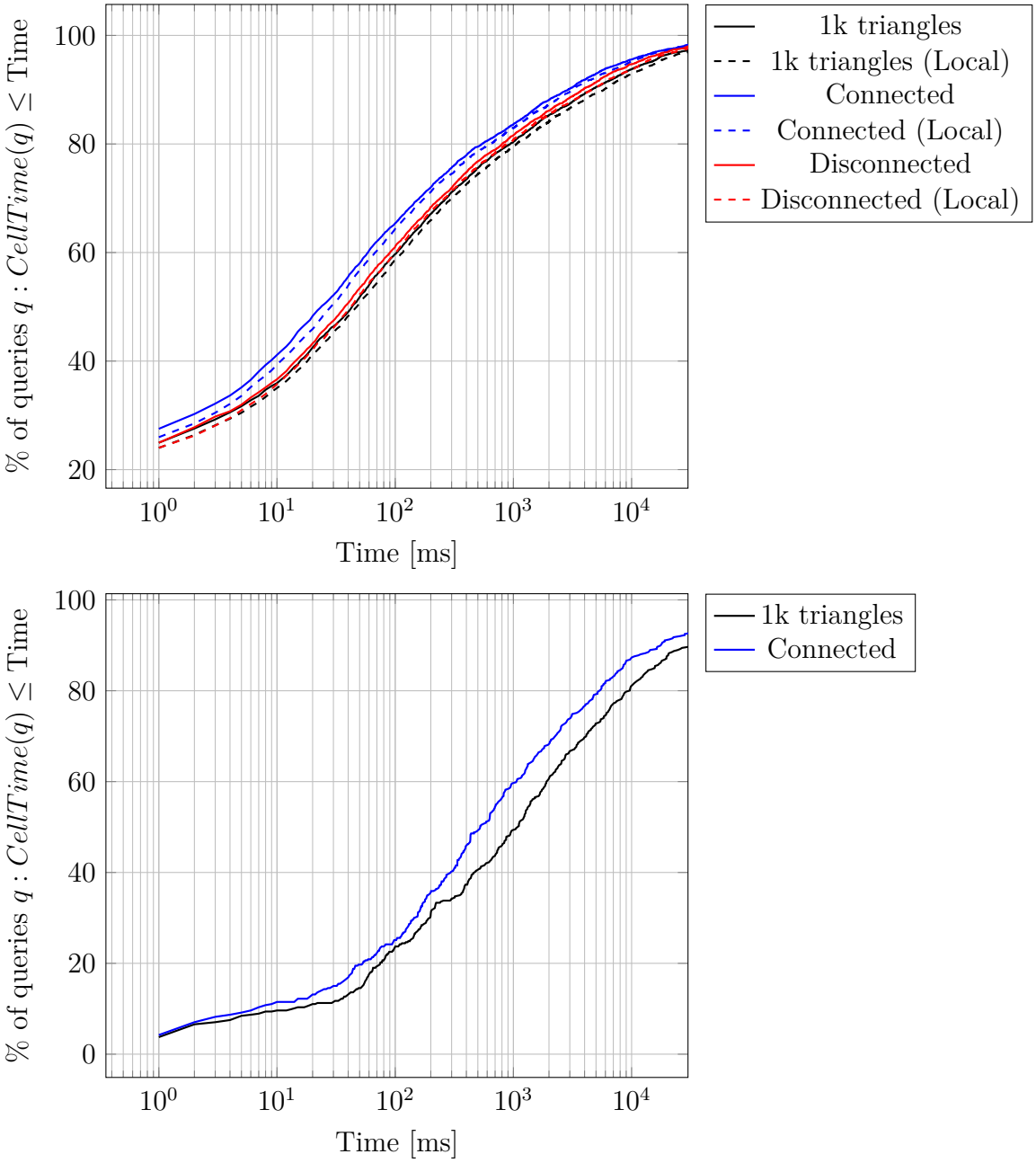


Figure 8.1: Cumulative distribution function to compute the cells of a query, standardized to the number of cells of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

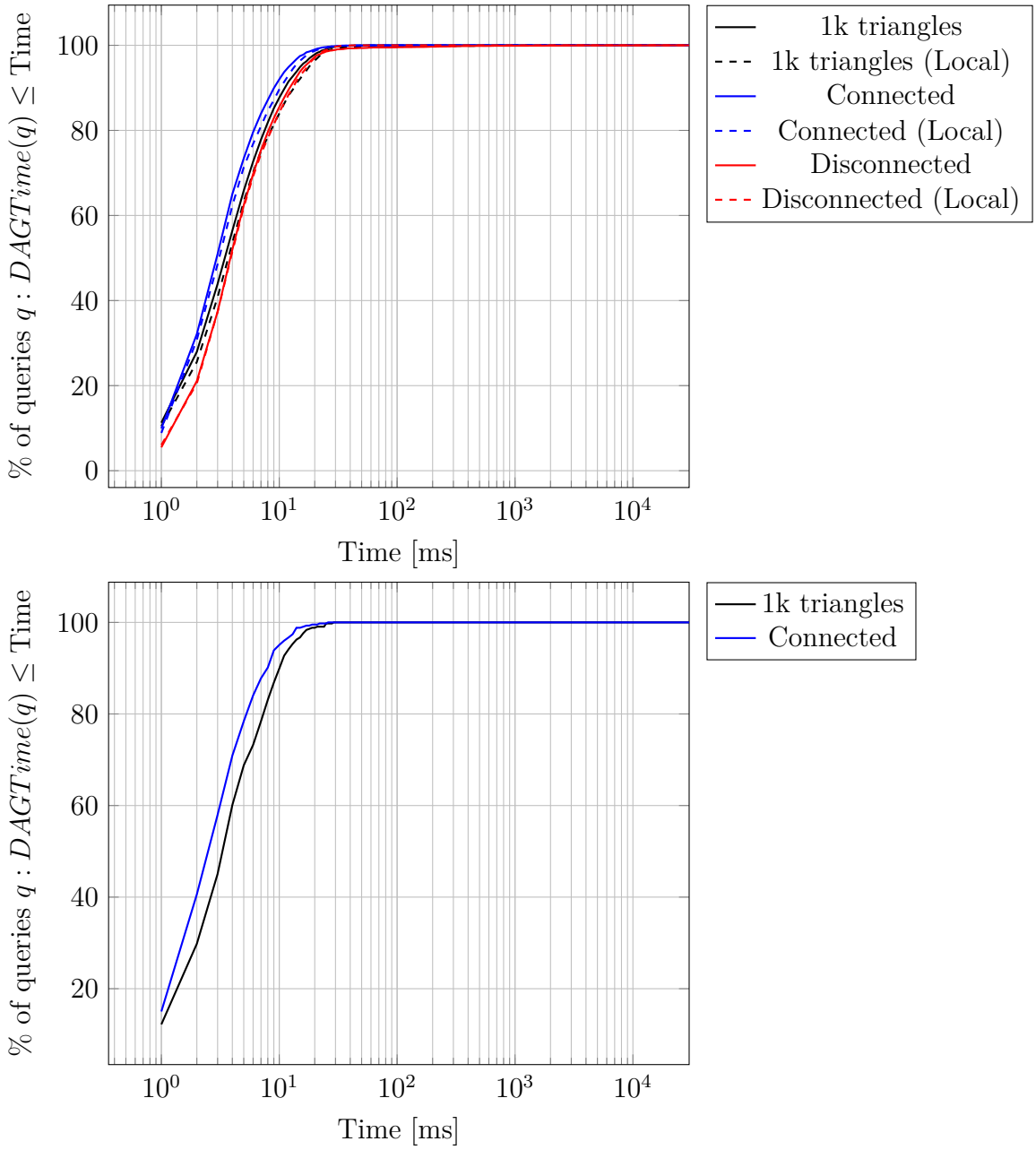


Figure 8.2: Cumulative distribution function to compute the subgraph of a query, standardized to the number of cells of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

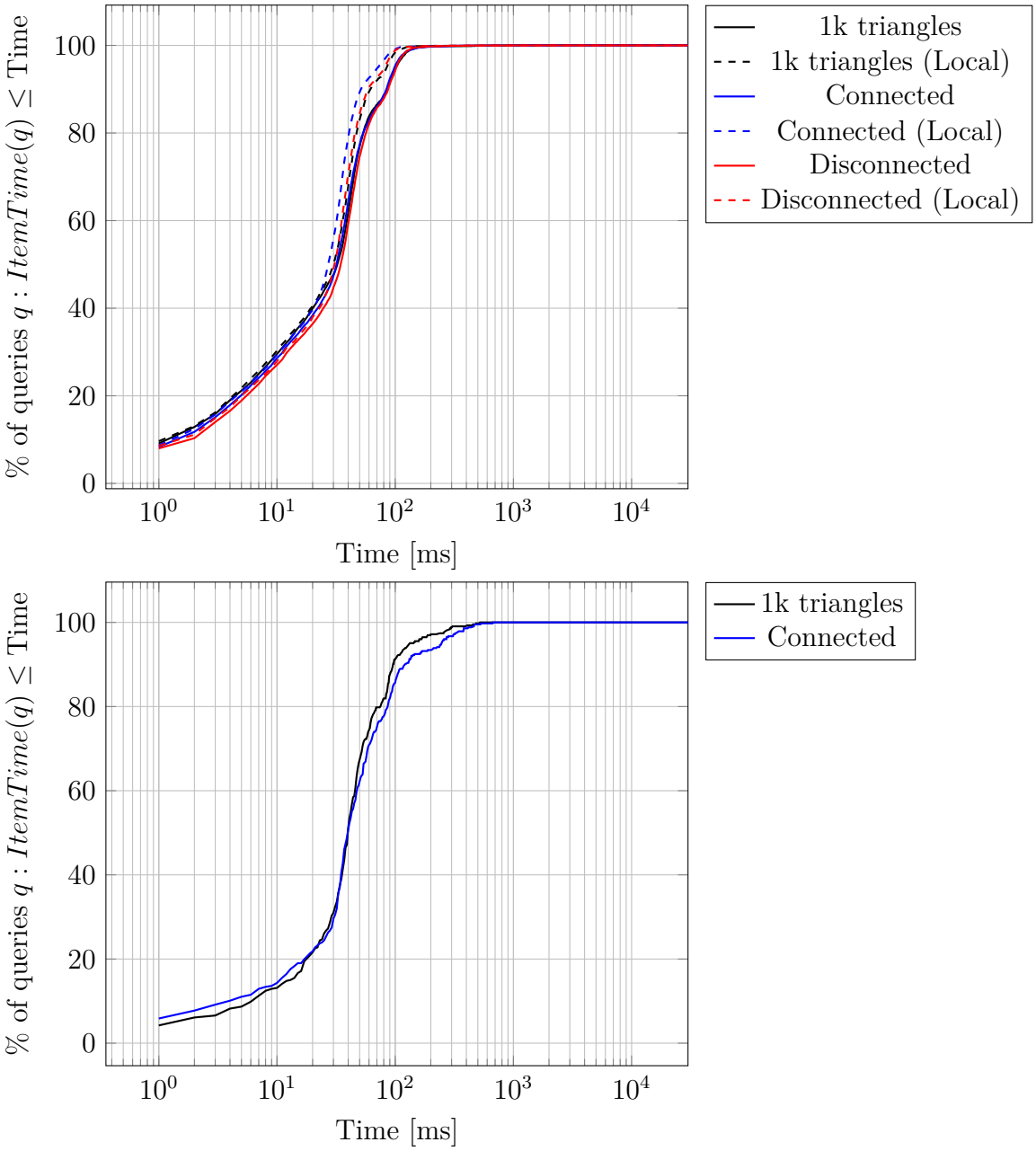


Figure 8.3: Cumulative distribution function to compute all items of a query, standardized to the number of cells of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

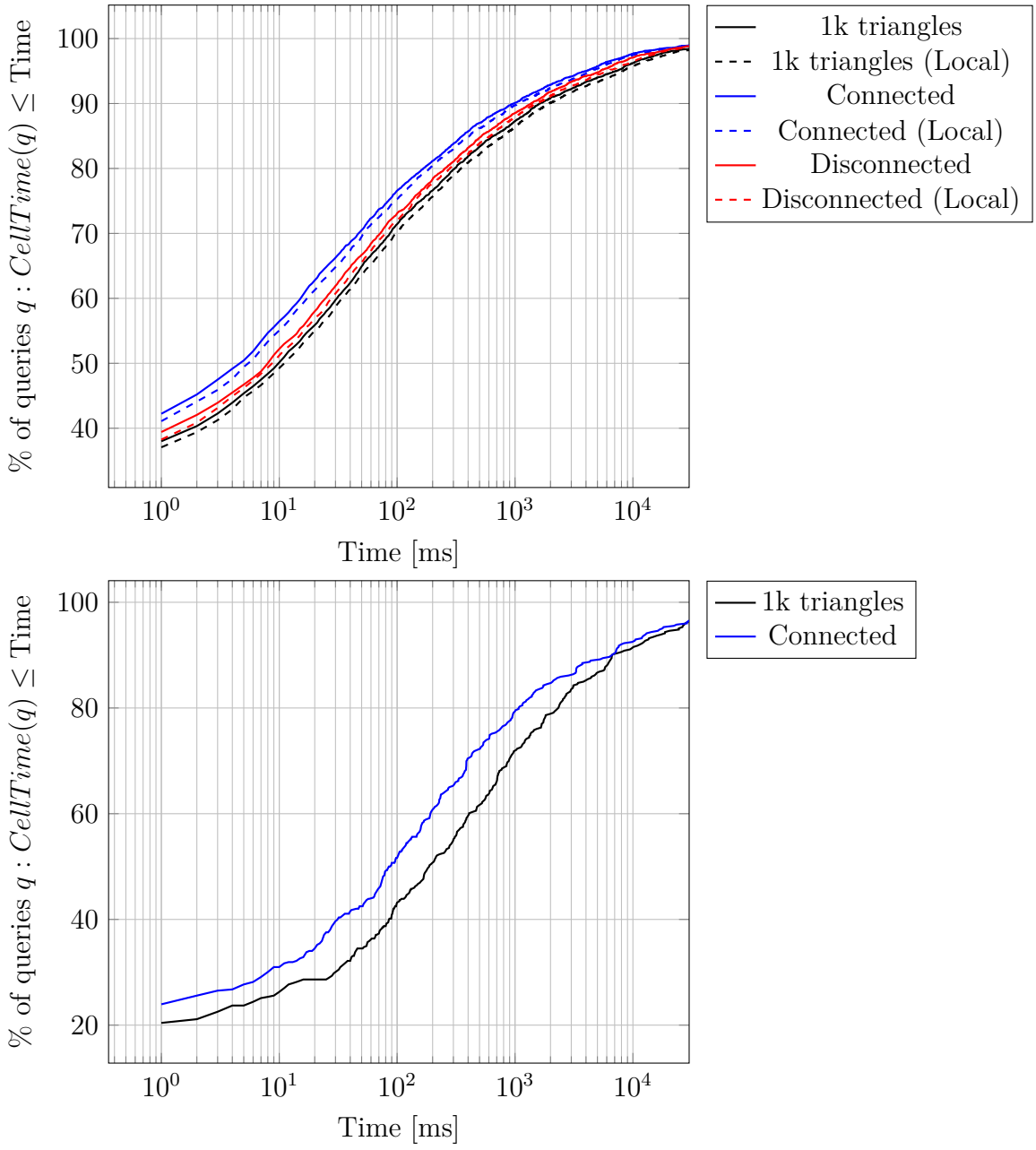


Figure 8.4: Cumulative distribution function to compute the cells of a query, standardized to the number of items of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

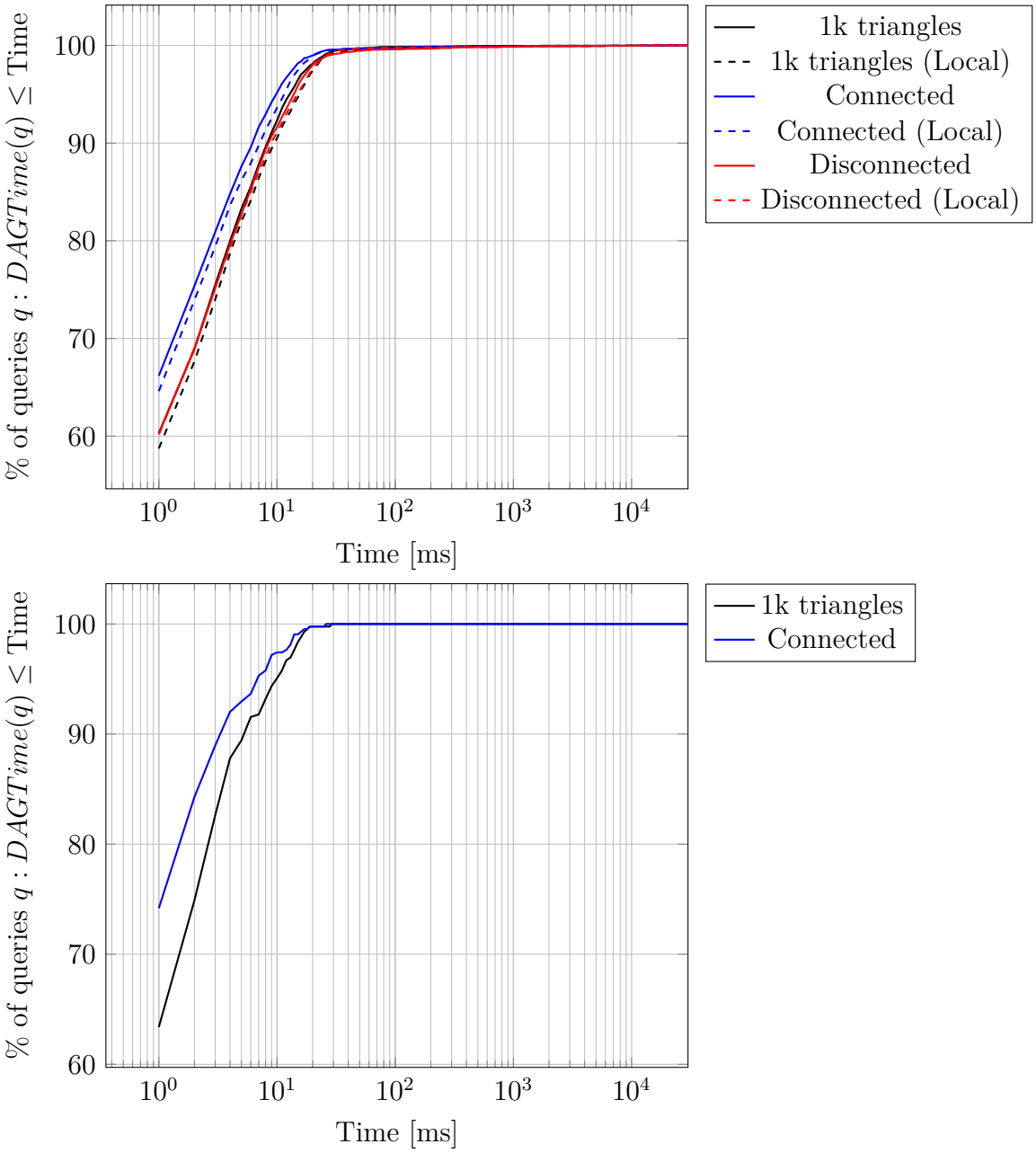


Figure 8.5: Cumulative distribution function to compute the subgraph of a query, standardized to the number of items of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

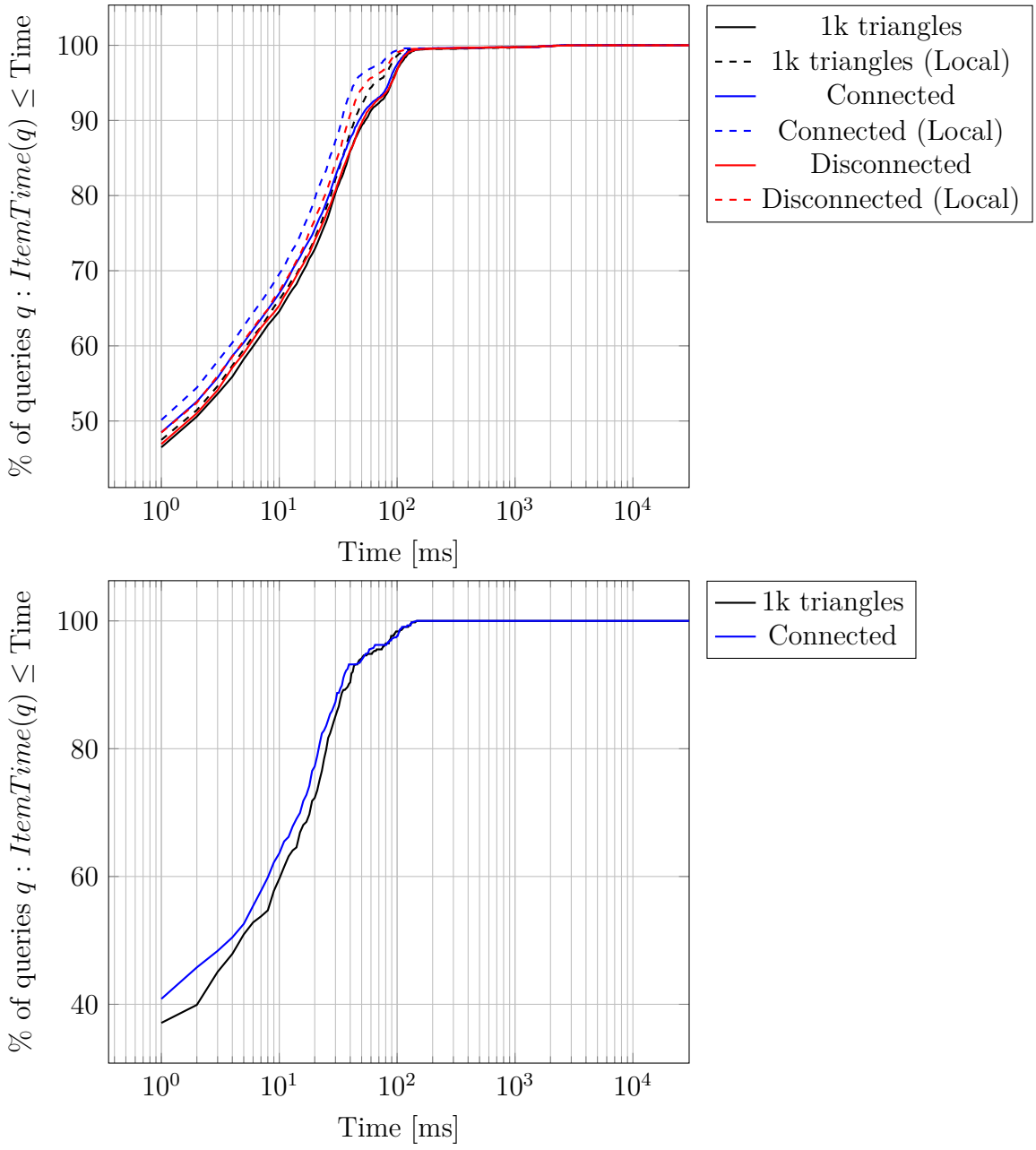


Figure 8.6: Cumulative distribution function to compute all items of a query, standardized to the number of items of the result, showing the impact of cell refinement and local vs. global item ids. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

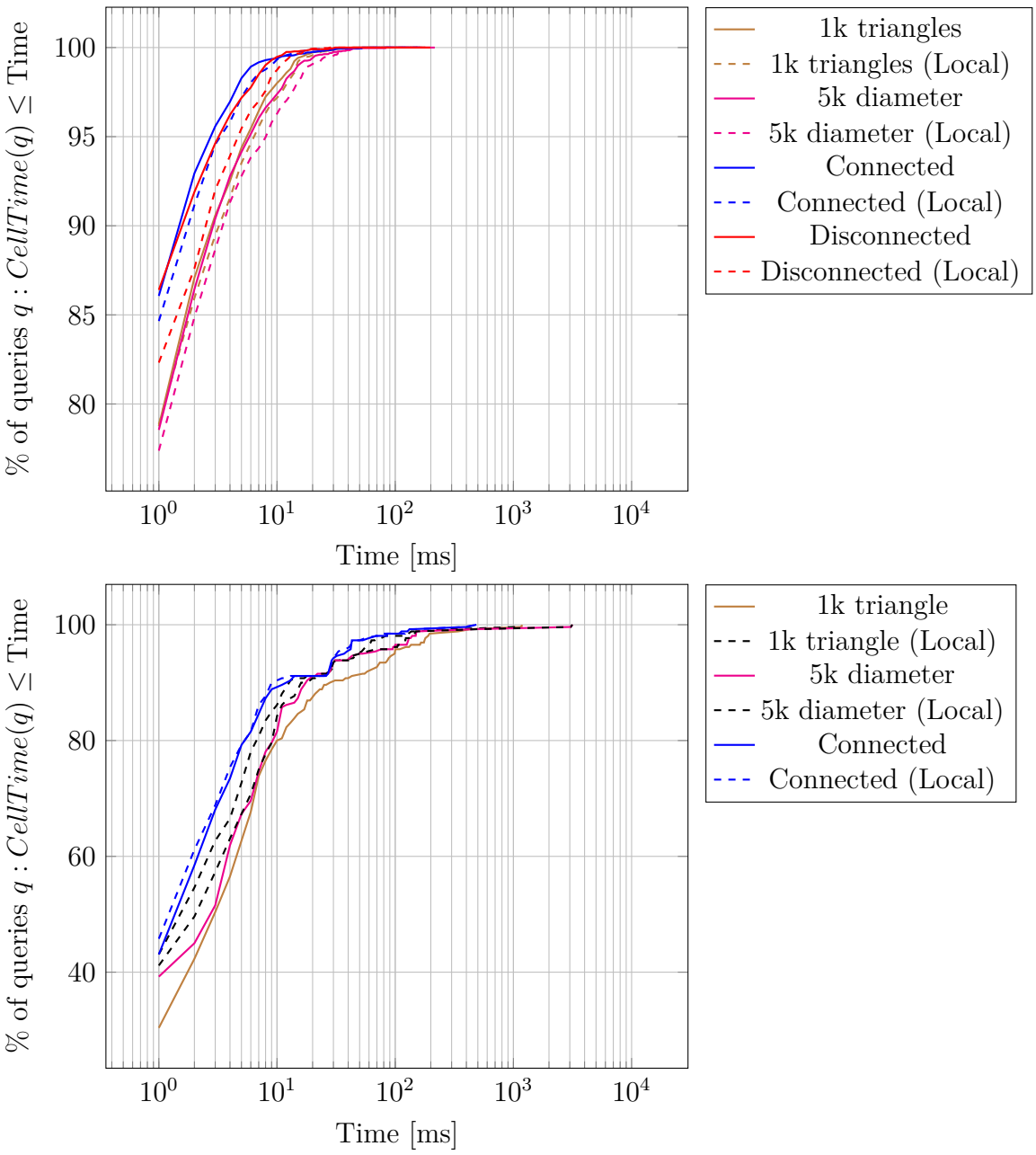


Figure 8.7: Cumulative distribution function to compute the cells of a query showing the impact of cell refinement and local vs. global item ids for the Baden-Württemberg data set using multiple threads. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

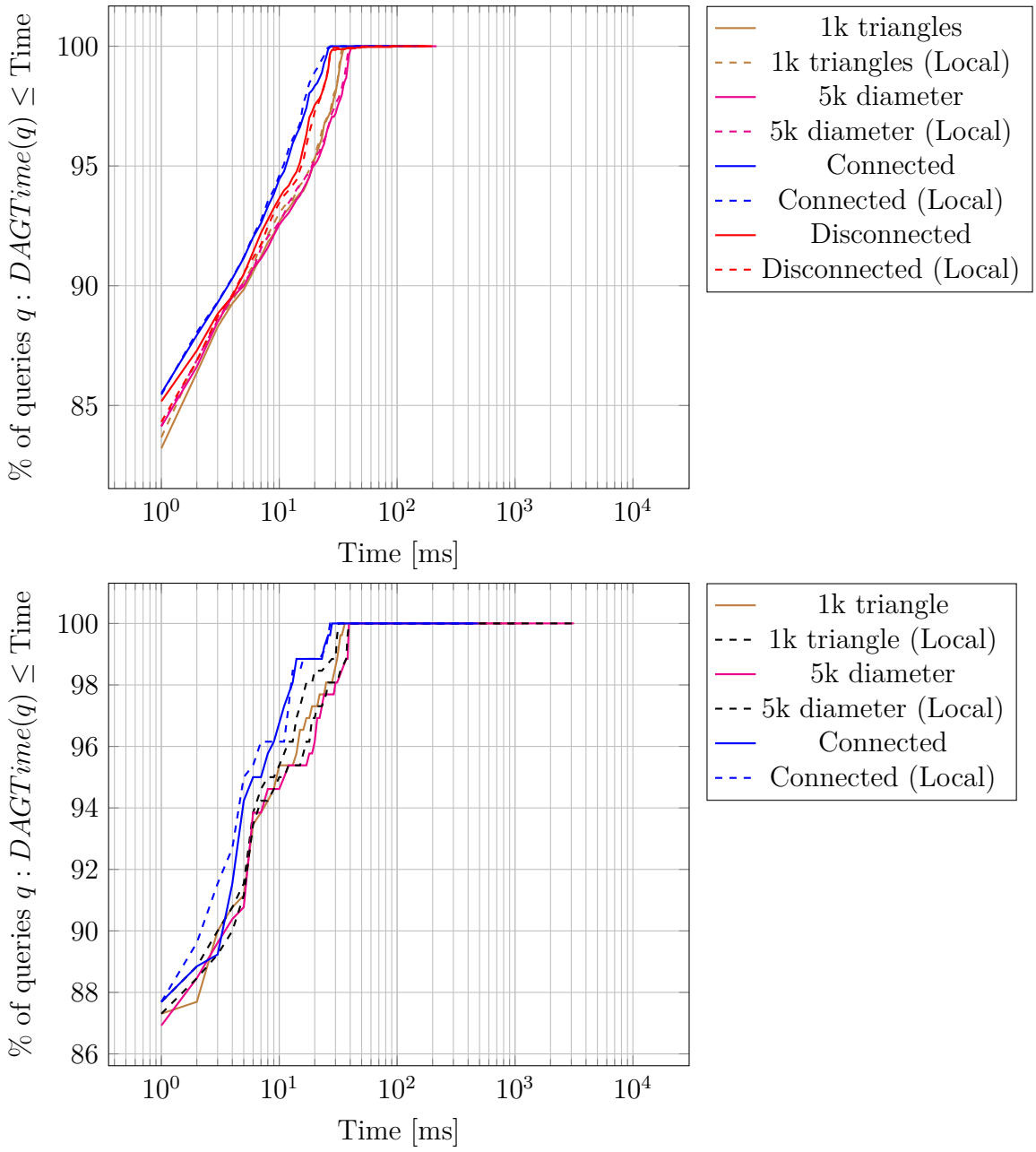


Figure 8.8: Cumulative distribution function to compute the subgraph of a query showing the impact of cell refinement and local vs. global item ids for the Baden-Württemberg data set using multiple threads. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

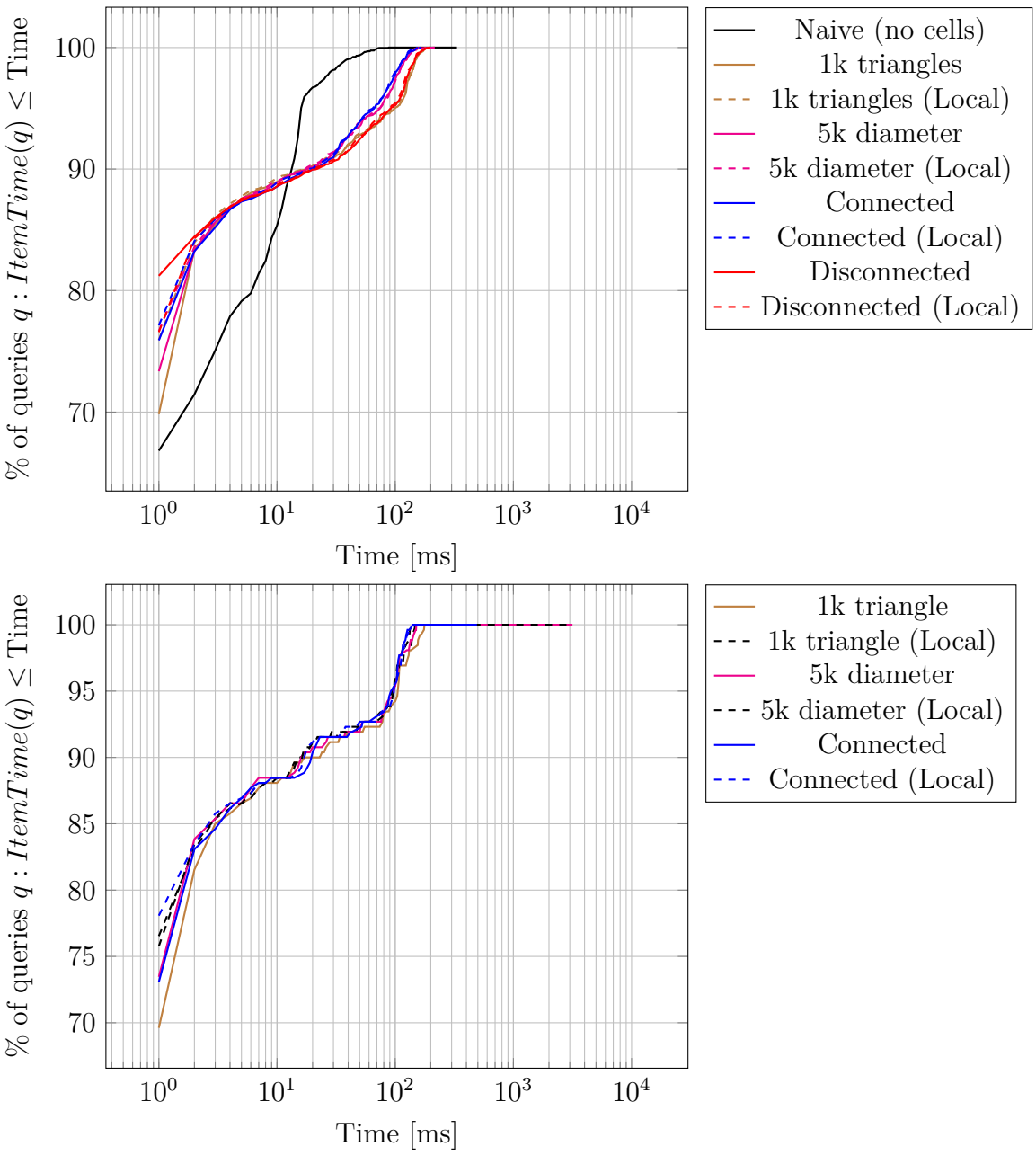


Figure 8.9: Cumulative distribution function to compute all items of a query showing the impact of cell refinement and local vs. global item ids for the Baden-Württemberg data set using multiple threads. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

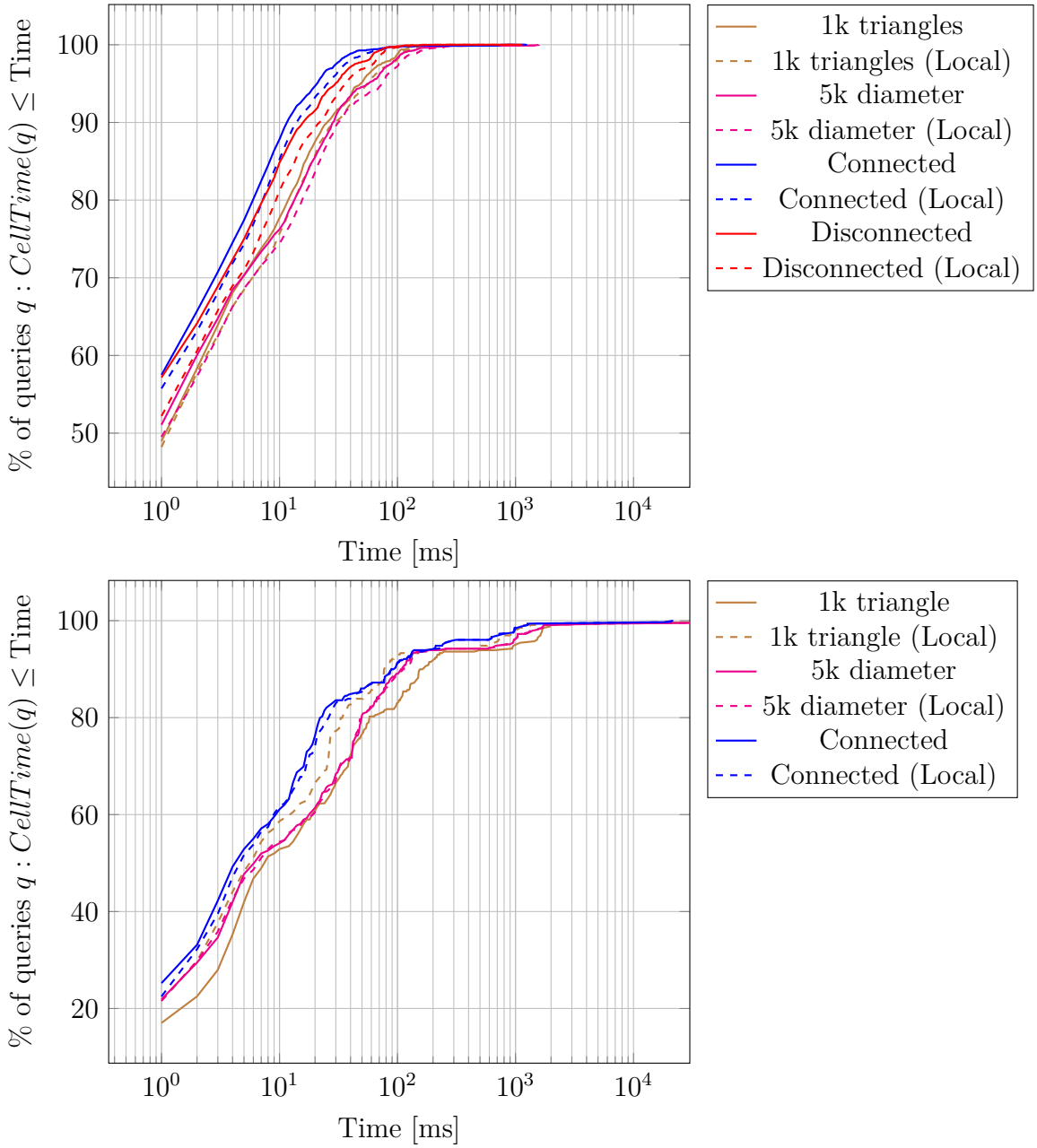


Figure 8.10: Cumulative distribution function to compute the cells of a query showing the impact of cell refinement and local vs. global item ids for the Germany data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

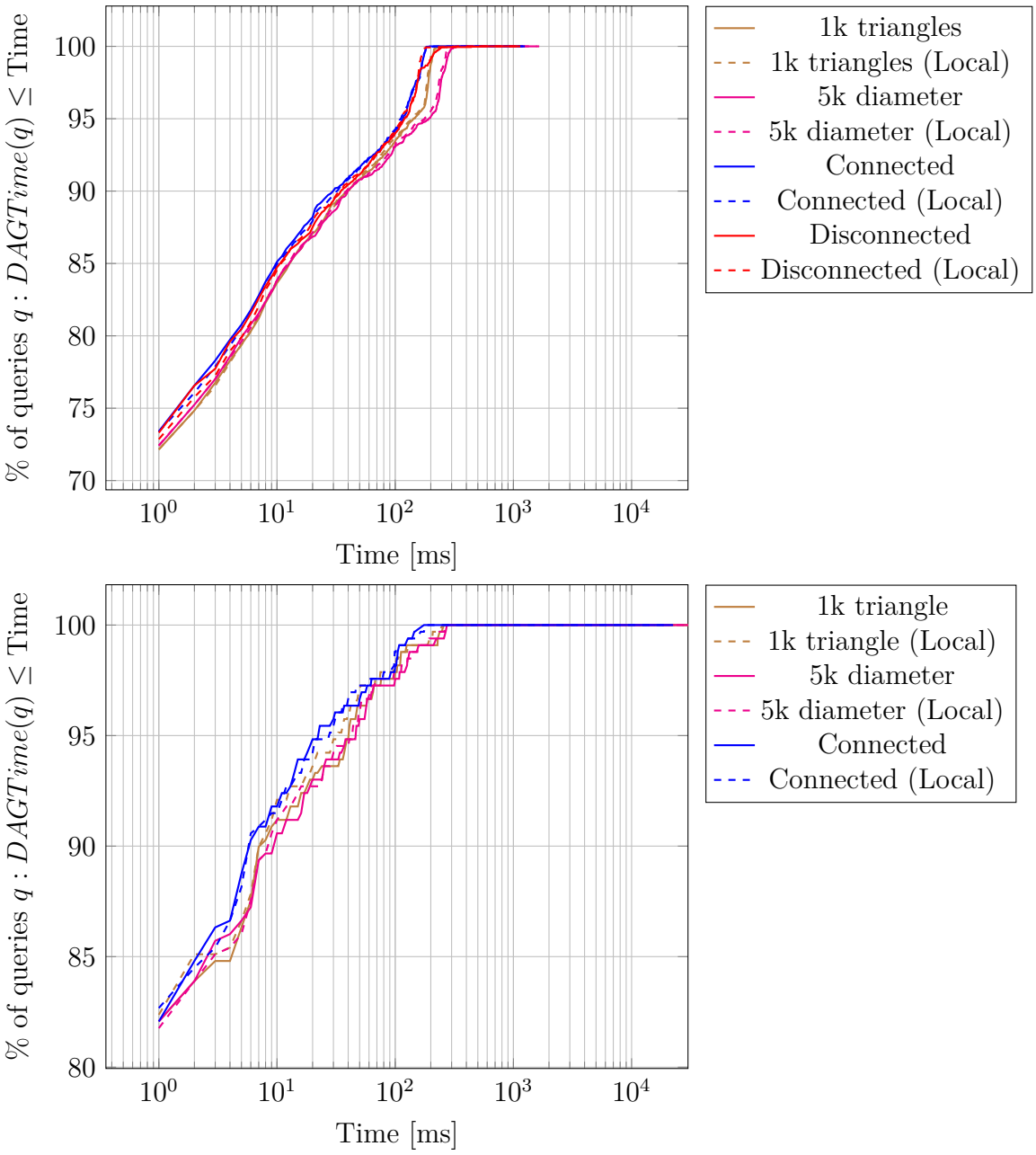


Figure 8.11: Cumulative distribution function to compute the subgraph of a query showing the impact of cell refinement and local vs. global item ids for the Germany data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

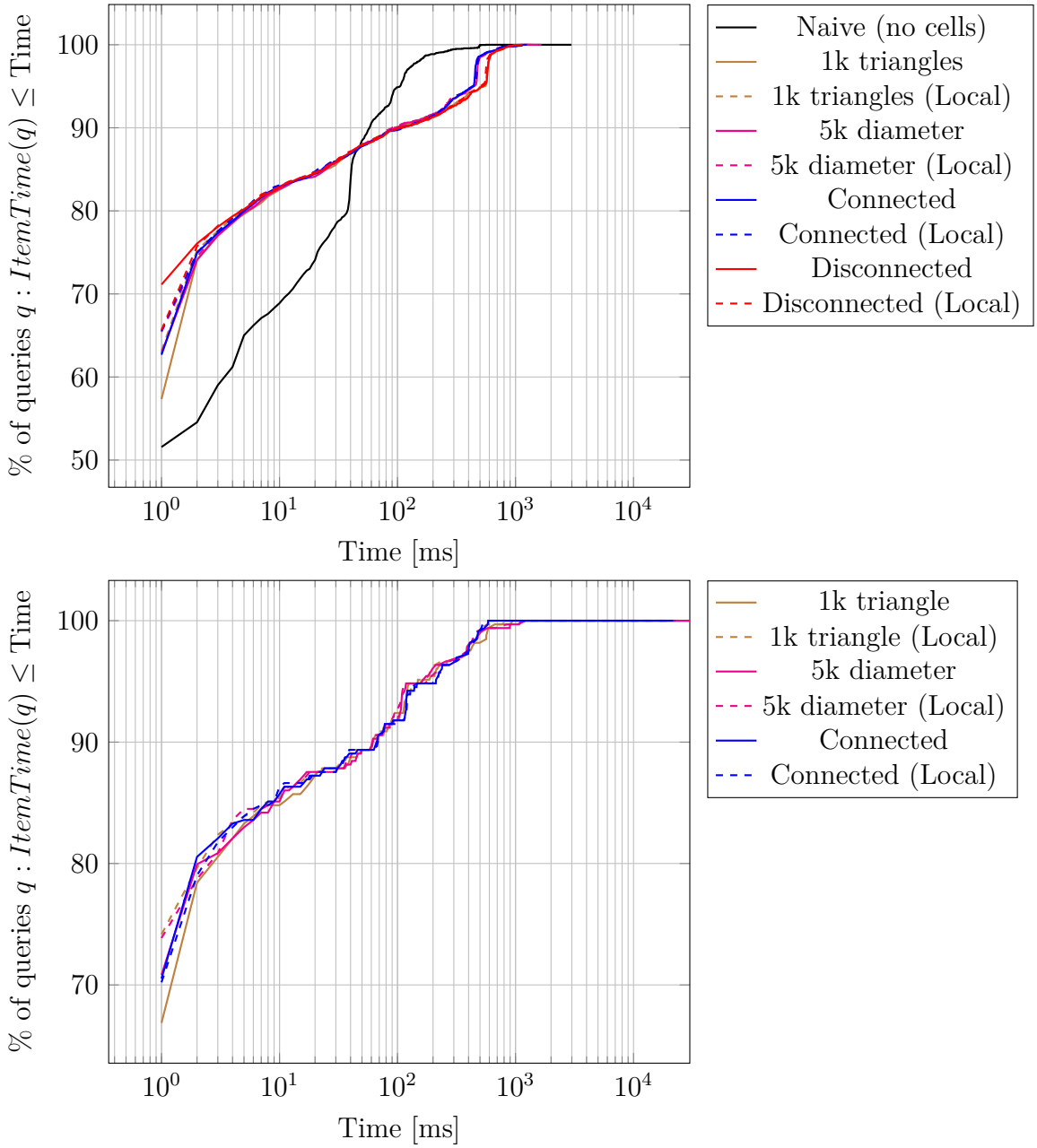


Figure 8.12: Cumulative distribution function to compute all items of a query showing the impact of cell refinement and local vs. global item ids for the Germany data set. The pictures at the top show the cdf for the textual website queries whereas the pictures at the bottom show the cdf for the spatial website queries.

Notes

1. From our paper [15]
2. From our paper [16]
3. Based on our paper [16]
4. Derived from our paper [15]
5. Figure is part of our paper [15]
6. Derived from our paper [15]
7. Figure is part of our paper [15]
8. Figure is part of our paper [15]
9. Section derived from our paper [15]
10. Figure is part of our paper [15]
11. Figure is part of our paper [15]
12. Figure is part of our paper [15]
13. Image is part of our paper [15]

Bibliography

[1] Apache Lucene Project, 2016.

[2] OpenStreetMap Project. www.openstreetmap.org, 2017.

[3] Specification of the OpenStreetMap Protocolbuffer Binary Format. https://wiki.openstreetmap.org/wiki/PBF_Format, 2017.

[4] Statistics for www.openstreetmap.org. stats.openstreetmap.org/cgi-bin/awstats.pl?config=www.openstreetmap.org, 2017.

[5] Geofabrik Website. https://www.geofabrik.de, 2018.

[6] A. I. Abdelmoty and C. B. Jones. Towards Maintaining Consistency of Spatial Databases. In *Proc. 6th Int. Conference on Information and Knowledge Management*, CIKM '97, 1997.

[7] B. Adams, G. McKenzie, and M. Gahegan. Frankenplace: Interactive Thematic Mapping for Ad Hoc Exploratory Search. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 12–22, 2015.

[8] S. Babicki, D. Arndt, A. Marcu, Y. Liang, J. R. Grant, A. Maciejewski, and D. S. Wishart. Heatmapper: web-enabled heat mapping for all. *Nucleic Acids Research*, 44(Webserver-Issue):W147–W153, 2016.

[9] T. Bagg. Resultatsclustering und einfache POI Exploration für OSCAR.

[10] D. Bahrdt. Osmfind: Fast Textual Search on OSM Data – on Smartphones and Servers. In *Proc. 2nd ACM SIGSPATIAL Int. Workshop on Mobile Geographic Information Systems*, MobiGIS '13, 2013.

[11] D. Bahrdt. Oscar experiments data. theogit.fmi.uni-stuttgart.de/bahrdtdl/oscar-experiments-data, 2018.

[12] D. Bahrdt. Oscar experiments source code. www.github.com/dbahrdt/oscar-experiments, 2018.

[13] D. Bahrdt, M. Becher, S. Funke, F. Krumpe, A. Nusser, M. Seybold, and S. Storandt. Growing Balls in $\mathbb{R}^d$. In *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.*, pages 247–258, 2017.

[14] D. Bahrdt and S. Funke. OSCAR: OpenStreetMap Planet at Your Fingertips via OSm Cell ARrangements. In *Web Information Systems Engineering - WISE 2015 - 16th International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part I*, pages 153–168, 2015.

[15] D. Bahrdt, S. Funke, R. Gelhausen, and S. Storandt. Searching OSM Planet with Context-Aware Spatial Relations. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 70:1–70:4, 2017.

[16] D. Bahrdt and M. P. Seybold. Rational Points on the Unit Sphere: Approximation Complexity and Practical Constructions. *CoRR*, abs/1707.08549, 2017.

[17] D. Bahrdt and M. P. Seybold. Rational Points on the Unit Sphere: Approximation Complexity and Practical Constructions. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, pages 29–36, 2017.

[18] D. Bahrdt and M. P. Seybold. Libdts2 library on GitHub. `www.github.com/fmi-alg/libdts2`, 2017.

[19] D. Bahrdt and M. P. Seybold. Libratss library on GitHub. `www.github.com/fmi-alg/libratss`, 2017.

[20] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[21] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*, pages 364–371, 2006.

[22] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.*, pages 322–331, 1990.

[23] A. Beilis. Cppcms website. www.cppcms.com, 2018.

[24] P. Boldi and S. Vigna. MG4J at TREC 2005. In *Proc. 14th Text REtrieval Conference, TREC, Gaithersburg, Maryland, USA, November 15-18, 2005*, 2005.

[25] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, RFC Editor, December 2017.

[26] K. Q. Brown. Voronoi Diagrams from Convex Hulls. *Inf. Process. Lett.*, 9(5):223–228, 1979.

[27] K. Buchin, V. Kusters, B. Speckmann, F. Staals, and B. Vasilescu. A Splitting Line Model for Directional Relations. In *Proc. 19th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, 2011.

[28] C. Burnikel, S. Funke, and M. Seel. Exact Geometric Predicates Using Cascaded Computation. In *Proc. of Sympos. on Comput. Geom.*, 1998.

[29] J. Canny, B. Donald, and E. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. of Sympos. on Comput. Geom.*, 1992.

[30] M. Caroli, P. M. M. de Castro, S. Loriot, O. Rouiller, M. Teillaud, and C. Wormser. Robust and Efficient Delaunay Triangulations of Points on Or Close to a Sphere. In *Proc. of Sympos. on Exp. Alg.*, pages 462–473, 2010.

[31] M. Catena, C. Macdonald, and I. Ounis. On Inverted Index Compression for Search Engine Efficiency. In *Advances in Information Retrieval - 36th European Conference on IR Research, ECIR 2014, Amsterdam, The Netherlands, April 13-16, 2014. Proceedings*, pages 359–371, 2014.

[32] L. P. Chew. Constrained Delaunay Triangulations. In *Proc. of Sympos. on Comput. Geom.*, pages 215–222, 1987.

[33] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *PVLDB*, 2(1):337–348, 2009.

[34] P. M. M. de Castro, F. Cazals, S. Loriot, and M. Teillaud. Design of the CGAL 3D Spherical Kernel and application to arrangements of circles on a sphere. *Comput. Geom.*, 42(6-7):536–550, 2009.

[35] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second International Conference on Web Search and Web Data Mining, WSDM 2009, Barcelona, Spain, February 9-11, 2009*, page 1, 2009.

[36] P. Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM*, 21(2):246–260, 1974.

[37] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, 21(2):194–203, 1975.

[38] K. V. et al. Cap'n Proto. `www.capnproto.org/index.html`, 2018.

[39] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword Search on Spatial Databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 656–665, 2008.

[40] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000.

[41] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[42] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[43] A. Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2018.

[44] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[45] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 319–333, 2008.

[46] GNU Project. Gnu time. `https://www.gnu.org/software/time`, 1999.

[47] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 370–379, 1998.

[48] M. T. Goodrich, L. J. Guibas, J. Hershberger, and P. J. Tanenbaum. Snap Rounding Line Segments Efficiently in Two and Three Dimensions. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry, Nice, France, June 4-6, 1997*, pages 284–293, 1997.

[49] R. K. Goyal. *Similarity Assessment for Cardinal Directions Between Extended Spatial Objects*. PhD thesis, University of Maine, 2000. AAI9972143.

[50] R. K. Goyal and M. J. Egenhofer. Similarity of Cardinal Directions. In *Proc. 7th Int. Symposium on Advances in Spatial and Temporal Databases*, SSTD '01, 2001.

[51] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[52] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000.

[53] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[54] O. Groß. osmpbf library on GitHub. `github.com/inphos42/osmpbf`, 2017.

[55] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57, 1984.

[56] D. Halperin and E. Packer. Iterated snap rounding. *Comput. Geom.*, 23(2):209–225, 2002.

[57] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press., 1954.

[58] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) Systems. In *19th International Conference on Scientific and Statistical Database Management, SSDBM 2007, 9-11 July 2007, Banff, Canada, Proceedings*, page 16, 2007.

[59] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance Evaluation of Main-Memory R-tree Variants. In *Advances in Spatial and Temporal Databases, 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 24-27, 2003, Proceedings*, pages 10–27, 2003.

[60] U. T. Inc. *H3: Uber's Hexagonal Hierarchical Spatial Index*.

[61] D. W. Jacobsen, M. Gunzburger, T. Ringler, J. Burkardt, and J. Peterson. Parallel algorithms for planar and spherical Delaunay construction with an application to centroidal Voronoi tessellations. *Geosci. Model Dev.*, 6(4), 2013.

[62] W. Jurkat, W. Kratz, and A. Peyerimhoff. On best two-dimensional Dirichlet-approximations and their algorithmic calculation. *Mathematische Annalen*, 244(1), 1979.

[63] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 500–509, 1994.

[64] D. G. Kirkpatrick. Optimal Search in Planar Subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[65] D. Kleinbock and K. Merrill. Rational approximation on spheres. *Israel Journal of Mathematics*, 209(1):293–322, 2015.

[66] A. Klippel, C. Dewey, M. Knauff, K.-F. Richter, D. R. Montello, C. Freksa, and E.-A. Loeliger. Direction Concepts in Wayfinding Assistance Systems. In J. Baus, C. Kray, and R. Porzel, editors, *Proc. Workshop on Artificial Intelligence in Mobile Systems (AIMS'04)*, number Memo 84 in SFB 378, Saarbrücken, 2004.

[67] C. Kumar, B. Poppinga, D. Haeuser, W. Heuten, and S. Boll. Assessing end-user interaction for multi-criteria local search with heatmap and icon-based visualizations. In *MapInteract 2013, Proceedings of the 1st ACM SIGSPATIAL International Workshop on MapInteraction, November 5th, 2013, Orlando, Florida, USA*, pages 13–19, 2013.

[68] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137, 2012.

[69] D. Lemire, N. Kurz, and C. Rupp. Stream VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.*, 130:1–6, 2018.

[70] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[71] M. Levene. Search Engines: Information Retrieval in Practice. *Comput. J.*, 54(5), 2011.

[72] C. Li, S. Pion, and C. Yap. Recent progress in exact geometric computation. *J. Log. Algebr. Program.*, 64(1):85–111, 2005.

[73] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou. Spatial Approximate String Search. *IEEE Trans. Knowl. Data Eng.*, 25(6):1394–1409, 2013.

[74] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. L. Lee, and X. Wang. IR-Tree: An Efficient Index for Geographic Document Search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[75] J. J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. MacDonald, and S. Vigna. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *Proc. 38th Europ. Conference on IR (ECIR)*, 2016.

[76] J. Liouville. Sur des classes très-étendues de quantités dont la valeur n'est ni algébrique, ni même réductible à des irrationalles algébriques. *J. Math. pures et app.*, 16:133–142, 1851.

[77] S. Makolli. Graphical Clustering of OSCAR Result Sets.

[78] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA.*, pages 319–327, 1990.

[79] G. Marsaglia. Choosing a point from the surface of a sphere. *The Annals of Mathematical Statistics*, 43(2), 1972.

[80] P. Matsakis and L. Wendling. A New Way to Represent the Relative Position Between Areal Objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(7), July 1999.

[81] T. Mendel. Area-Preserving Subdivision Simplification with Topology Constraints: Exactly and in Practice. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018.*, pages 117–128, 2018.

[82] J. I. Munro and S. Wild. Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, pages 63:1–63:16, 2018.

[83] National Imagery and Mapping Agency. Department of Defense World Geodetic System 1984: its definition and relationships with local geodetic systems. Technical Report TR8350.2, National Imagery and Mapping Agency, St. Louis, MO, USA, Jan. 2000.

[84] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[85] I. Niven. *Irrational Numbers*, volume 11. Math. Assoc. of America, 1 edition, 1985.

[86] T. Participants and Organizers. Text REtrieval Conference (TREC) Data. `https://trec.nist.gov/data.html`, 2018.

[87] A. Pethő, M. E. Pohst, and C. Bertók. On multidimensional Diophantine approximation of algebraic numbers. *J. Number Theory*, 171, 2017.

[88] D. J. Peuquet and C.-X. Zhang. An Algorithm to Determine the Directional Relationship Between Arbitrarily-shaped Polygons in the Plane. *Pattern Recogn.*, 20(1), Jan. 1987.

[89] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte Decoding. *CoRR*, abs/1503.07387, 2015.

[90] F. Prill and G. Zängl. A Compact Parallel Algorithm for Spherical Delaunay Triangulations. In *Proc. Conf. on Parallel Processing and Appl. Math.*, 2015.

[91] T. C. Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.11 edition, 2017.

[92] T. C. Project. Cross Compilation of CGAL for Android, 2017.

[93] M. B. J. Purss. Discrete Global Grid Systems Abstract Specification. Technical report, Open Geospatial Consortium, 2017.

[94] M. B. J. Purss, R. Gibb, F. F. Samavati, P. Peterson, and J. Ben. The OGC® Discrete Global Grid System core standard: A framework for rapid geospatial integration. In *2016 IEEE International Geoscience and Remote Sensing Symposium, IGARSS 2016, Beijing, China, July 10-15, 2016*, pages 3610–3613, 2016.

[95] R. S. Purves, P. D. Clough, C. B. Jones, M. H. Hall, and V. Murdock. Geographic Information Retrieval: Progress and Challenges in Spatial Search of Text. *Foundations and Trends in Information Retrieval*, 12(2-3):164–318, 2018.

[96] R. J. Renka. Algorithm772 STRIPACK Delaunay Triangulation and Voronoi Diagram on the Surface of a Sphere. *Trans. Math. Softw.*, 23(3), 1997.

[97] T. J. Rivlin. *The Chebyshev Polynomials: From Approximation Theory to Algebra and Number Theory*. Tracts in Pure & Applied Mathematics. Wiley, 1974.

[98] A. Saalfeld. Delaunay Triangulations and Stereographic Projections. *Cartography and Geographic Information Science*, 26(4):289–296, 1999.

[99] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, pages 410–421, 2000.

[100] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[101] E. Schmutz. Rational points on the unit sphere. *Central European Journal of Mathematics*, 6(3), 2008.

[102] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518, 1987.

[103] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 317–326, 2011.

[104] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the Sphere with the Hierarchical Triangular Mesh. *CoRR*, abs/cs/0701164, 2007.

[105] H. Tachii. On Jacobi-Perron algorithm. In *Proceedings of the Japan Academy, Series A, Mathematical Sciences*, volume 70, pages 317–322. The Japan Academy, 1994.

[106] A. Trotman, X. Jia, and M. Crane. Towards an efficient and effective search engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, 2012.

[107] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual Indexing for Geographical Search on the Web. In *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, pages 218–235, 2005.

[108] W. van Oortmerssen et al. Flatbuffers. `www.google.github.io/flatbuffers`, 2018.

[109] S. Vigna. Quasi-succinct Indices. In *Proc. 6th ACM Int. Conf. on Web Search and Data Mining 2013*, WSDM '13, 2013.

[110] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. AP-Tree: efficiently support location-aware Publish/Subscribe. *VLDB J.*, 24(6):823–848, 2015.

[111] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, Mar. 2006.

[112] S. S.-T. Yau and Q.-L. Zhang. On Completeness of Reasoning about Planar Spatial Relationships in Pictorial Retrieval Systems. *Commun. Inf. Syst.*, 4(3), 2004.

[113] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted Linear Quadtree: Efficient Top K Spatial Keyword Search. *IEEE Trans. Knowl. Data Eng.*, 28(7):1706–1721, 2016.

[114] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Ma. Hybrid index structures for location-based web search. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 155–162, 2005.

[115] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59, 2006.