DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Spatio-Textual Joins

Maximilian Bandle

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Spatio-Textual Joins

Effiziente Spatio-Textual Joins

Author:	Maximilian Bandle
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisor:	Andreas Kipf, M.Sc.
Submission Date:	November 15, 2018

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, October 24, 2018 Maximilian Bandle

Acknowledgments

I would like to express my gratitude to my advisor Andreas Kipf and my supervisor Prof. Kemper for the opportunity to work on this interesting topic. Andy provided useful comments and remarks through the learning process of this Master's Thesis while allowing me to develop the thesis in the way I wanted.

Furthermore, I would like to thank Ahmed Mahmood from Purdue University for giving me a workload to start my thesis and the code of his work on this topic to compare and learn from his research.

Also, I like to thank the people working at the chairs I3 and I25 for the nice time while working on the thesis. Especially Timo who shared his office with me and made good remarks on my plots and writing, Moritz who had some critical questions about correctness, Viktor who allowed me to use the server for doing the evaluation and Prof. Neumann for some remarks how to measure performance and memory consumption of code.

Special thanks to my family, especially my parents Frank and Birgit and my girlfriend Thuy Ngan who supported me throughout the whole process and gave me the time to study and write this thesis.

Many people, especially my friends have made valuable suggestions on my thesis. Without your ideas and corrections, the thesis would not be as good as it is now. Thanks to Thuy for proofreading and providing ideas throughout the whole process of writing. Thanks to Astrid and Caren for having a non-informatics view on this thesis, pointing out how to improve my writing. Thanks to Tobi who provided feedback in very early stages of writing and to Anna who pointed out some obscurities in my plots and my writing. Thanks to Alex who helped me to remove the small flaws which occurred during the writing.

Abstract

Increased day-to-day smartphone usage leads to a massive increase in geo-tagged data. Generally, many applications aim to process this spatio-textual data in real-time. For example, location-aware ad-targeting systems have to serve customized advertisements to millions of users as they continually alter their position. There are several optimized geospatial indexes with the Adaptive Cell Trie (ACT) as state-of-the-art index for fast probing. There further are several efficient spatio-textual indexes, for example the Frequency Aware Spatio-Textual Index (FAST). However, both options have shortcomings. ACT lacks textual support and FAST is not optimized consistently for modern hardware.

This Master's Thesis combines both approaches into a new optimized spatio-textual index. It first presents an overview of general techniques used in the field of spatio-textual indexing and critically examines the most efficient competitor FAST whose textual index is named Adaptive Keyword Index (AKI). AKI is reimplemented to optimize throughput and memory consumption to ultimately be a meaningful competitor for evaluation. Based on this new implementation, textual predicate support is added to the geo-spatial ACT increasing its throughput. Relatedly, the spatio-textual dataset used in the original evaluation of FAST did not provide a sound use-case. To tackle this issue, we present two new datasets which are derived from real-world points of interest (POI) and visitor data to provide a more realistic workload.

Finally, the optimized AKI and the textual ACT are compared with their respective state-of-the-art competitors in the experimental evaluation. The optimized AKI provides constant and fast probing performance and is more robust against specifics of the dataset than its competitors whilst ACT is capable of outperforming FAST by several orders of magnitude.

Kurzfassung

Durch die Verbreitung von Smartphones im Alltag stellen Menschen wesentlich mehr ortsbezogene Anfragen. Eine große Anzahl von Systemen versucht diese spatio-textuellen Joins in Echtzeit auszuführen um den Nutzern direkt Ergebnisse zu liefern, wie zum Beispiel ortsabhängige Werbung. Diese muss anhand der Position für Millionen von mobilen Nutzern angezeigt werden, um den Bedürfnissen der Werbekunden zu entsprechen.

Es gibt optimierte Index-Strukturen für Geodatan, wie der Adaptive Cell Trie (ACT) der im Moment die Daten am schnellsten verarbeiten kann. Außerdem existieren verschiedene spatio-textuelle Datenstrukuturen wie der Frequency Aware Spatio-Textual Index (FAST), der sich an die textuelle Selektivität anpasst. Jedoch haben beide Strukturen Nachteile, der ACT unterstützt keine textuellen Anfragen, während FAST nicht konsequent für moderne Hardware optimiert ist.

Diese Master-Arbeit kombiniert beide Ansätze in einen neuen optimierten Index für spatio-textuelle Joins. Hierfür wird zunächst ein Überblick über Techniken gegeben, die aktuell für spatio-textuelle Joins genutzt werden. Anschließend wird FAST, der effizienteste Mitbewerber, mit speziellem Augenmerk auf den textuellen Bestandteil, den Adaptive Keyword Index (AKI), untersucht. AKI wird neu implementiert um Geschwindigkeit und Speichereffizienz weiter zu verbessern. Aufbauend auf dieser Implementierung wird der ACT um textuelle Unterstützung ergänzt und dadurch auch beschleunigt. Des Weiteren sind die spatio-textuellen Datensets, die in der ursprünglichen Auswertung von FAST verwendet wurden, nicht besonders aussagekräftig. Deswegen werden zwei neue Datensets vorgestellt, die echte Orte und Besucherdaten als Basis verwenden und damit einen realistischen Datensatz für spatio-textuelle Joints darstellen.

Abschließend werden der optimierte AKI und der textuelle ACT mit ihren jeweiligen aktuellen Mitbewerbern experimentell verglichen. Der optimierte AKI erreicht hierbei eine konstant schnelle Leistung, während die anderen Indexe deutlich weniger robust gegen Änderungen der Daten sind. ACT ist bei sich nicht überschneidenden Polygonen mehrere Größenordnungen schneller als FAST.

Contents

Acknowledgments **v**

Abstract **vii**

Kurzfassung **ix**

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Use Case . . . 3
  - 1.2.1 Pollination . . . 3
  - 1.2.2 Advertisement . . . 4
- 1.3 Problem Definition . . . 5
  - 1.3.1 Geographical Containment . . . 6
  - 1.3.2 Textual Matching . . . 6
- 1.4 State-of-the-Art Methods . . . 6
  - 1.4.1 Spatial Indexes . . . 7
  - 1.4.2 Textual Subscription Indexes . . . 8
  - 1.4.3 Combination Methods . . . 9
- 1.5 Structure and Content of Thesis . . . 10

2 Building Blocks **11**
- 2.1 Adaptive Cell Trie . . . 11
  - 2.1.1 S2 . . . 11
  - 2.1.2 Cell-Trie . . . 12
  - 2.1.3 Payload Marker in Pointer . . . 12
- 2.2 Adaptive Keyword Index . . . 13
  - 2.2.1 Ranked-Key Inverted List . . . 13
  - 2.2.2 Ordered Keyword Trie . . . 14
  - 2.2.3 Combination . . . 16
- 2.3 Frequency-Aware Spatio-Textual Indexing . . . 17
  - 2.3.1 Combination of Spatial and Textual Properties . . . 17
  - 2.3.2 Shared Frequency Information . . . 18
  - 2.3.3 Spatial Sharing . . . 18
- 2.4 Various Methods . . . 19
  - 2.4.1 Keyword Dictionary . . . 19
  - 2.4.2 Bloom Filter . . . 19
  - 2.4.3 Hashing Algorithms . . . 20

3 Efficient Spatio-Textual Joins **23**
3.1 Adaptive Keyword Index 23
3.1.1 Node Type Reduction 23
3.1.2 Keyword Dictionary 24
3.1.3 Bloom Filter 24
3.1.4 Term Reduction 26
3.2 Extend ACT by Textual Discrimination 29
3.2.1 Prototype for Correctness 29
3.2.2 Pre-Selection by Bloom Filters 30
3.2.3 Replacing Large Lookup Tables by Textual Index Structures 31
3.2.4 Problems with Overlapping Regions 33

4 Data Generation **35**
4.1 Extruding Points to Regions 35
4.1.1 Individual Extruding 35
4.1.2 Domain Covering Polygons 36
4.2 Uniform Data 36
4.3 Naturally Distributed 37
4.3.1 OpenSubtitle Database 37
4.3.2 Foursquare Inspired 38
4.4 Yelp Dataset 38
4.4.1 Expanding Business Locations to Regions 39
4.4.2 Generating Visitor Locations 39
4.4.3 Generating Term Lists 41

5 Evaluation **43**
5.1 Datasets Overview 43
5.1.1 Uniform 43
5.1.2 Naturally Distributed 44
5.1.3 Yelp Dataset 45
5.1.4 Twitter Dataset 46
5.2 Experimental Setup 47
5.3 Subscription Indexes 48
5.3.1 Hash Table Comparison 48
5.3.2 Competitors 49
5.4 Spatio-Textual Indexes 51
5.4.1 Comparison of FAST Java with FAST C++ 51
5.4.2 Multithread Performance of C++ Implementation 53
5.4.3 Parametrized Uniform Dataset 54
5.4.4 Comparison of Textual ACT with FAST C++ 55

6 Conclusions **57**

List of Figures **59**

List of Tables **61**

Bibliography **63**

1 Introduction

In the last decade, the amount of geo-tagged data massively grew due to the wide-spread use of smartphones as shown in Figure 1.1. Nowadays, most people possess a smartphone which can geo-tag images, upload posts with location or just search for nearby Italian restaurants with credit card payment.

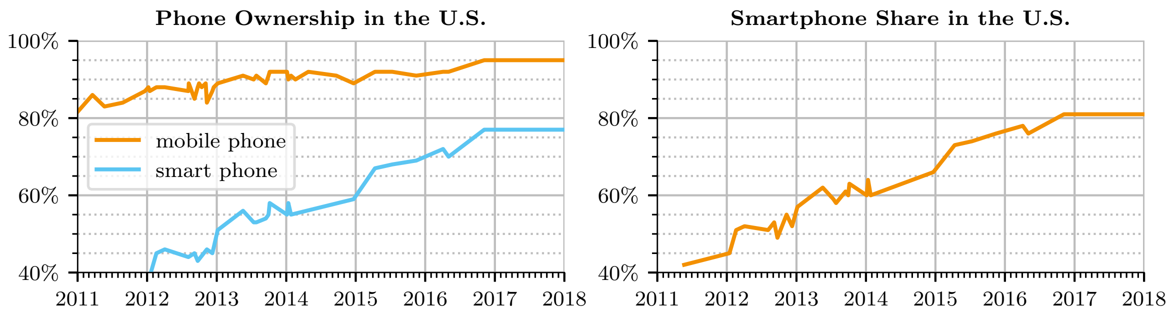


Figure 1.1: Mobile Phone Ownership in the U.S. (Data from [Pew18])

Most users unknowingly issue queries of this kind on a daily basis. This shows a general evolvement of the internet from a generic non-personal content delivery system to an individual experience. Most of the users have their own personalized newsfeeds, weather forecasts and request articles or information for their needs. They furthermore expect to get a tailored output when e.g. searching for restaurants. Thus, every response has to be customized in real-time. Due to the growing number of geotagged requests, special data structures which can handle both, spatial and textual selection have to be used.

These data structures are optimized for performing spatio-textual joins. They simultaneously check for geographical and textual predicates. Additional effort when processing data is required through the use of further processing steps. In general, the user is not interested in the region itself, he wants to see attached information, like the weather forecast.

Nowadays existing services process massive amounts of requests on a daily basis. Foursquare had more than 12 billion check-ins in total with a maximum of 9 million per day. Their database consists of 105 million places around the world [Fou18].

Yelp's mobile website is used monthly by about 72 million people, the app by 32 million users. Combined with the desktop users, more than 160 million reviews are generated and assigned to places. The development of the service's usage statistics is shown in Figure 1.2.

Thus, every application needs a fast, reliable and flexible backend which can handle the massive amount of newly generated data while handling spatio-textual requests.

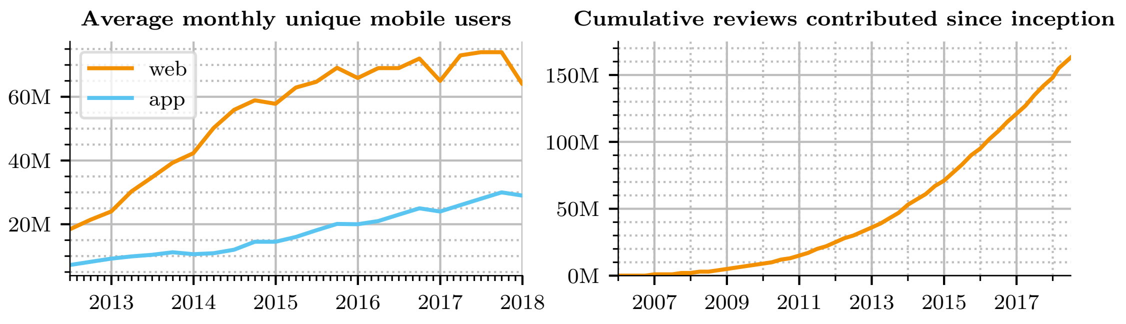


Figure 1.2: Various Yelp Usage Metrics (Data from [Yel18a])

1.1 Motivation

The *Adaptive Cell Trie* (ACT) is an efficient geo-spatial data-structure developed at TU München. It uses true hit filtering to avoid the time-consuming geometric computations and is optimized for modern hardware architectures. Currently, it is the fastest structure joining polygons with points on CPUs, especially when using SIMD instructions [Kip+18].

The *Frequency-Aware Spatio-Textual Index* (FAST) is an approach to speed up spatio-textual joins by using optimized index nodes to differentiate adaptively between frequent and non-frequent terms. It also uses novel approaches for balancing between spatial and textual selectivities. [MAA17; Mah18] The index originates from the distributed *Tornado* system developed by Mahmood et al. which focus on efficiently handling geo-spatial joins on several clusters partitioning the data. [Mah+17; Mah+15]

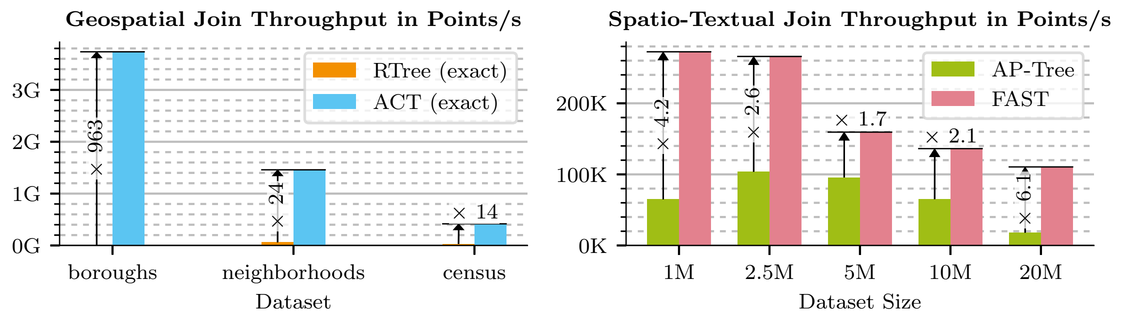


Figure 1.3: Comparison of FAST and ACT with their Respective Competitors (Data from [MAA17; Kip+18])

Both approaches are the fastest on their field as shown in Figure 1.3[1] and focus on different aspects. ACT focuses on raw performance for geospatial data processing on modern hardware. FAST mainly contributes to frequency aware text indexing. In combination, ideas of both approaches can lead to a new faster data structure for spatio-textual join processing on modern hardware.

As already stated, there is a demand to process geo-tagged data [Kip17]. Several systems

[1]The measurements for ACT And FAST were performed on different hardware setups. The Figure only illustrates the speed-up against the direct competitor.

exist which can either process spatial information or textual information. This problem is addressed in this thesis. The resulting data structure will combine the performance of ACT with the textual processing power of the FAST index to achieve a high throughput for spatio-textual data objects.

1.2 Use Case

In order to get used to the nomenclature when processing spatio-textual data, two examples of real-life tasks are given. The first example shows a scenario with single keywords, while the second example uses multiple keywords per entry. A formal problem definition is given in the following Section 1.3.

Each workload consists of polygons which have to be registered in the index and a stream of objects which are tested whether they match the region. Depending on the use case, either all matching regions are notified or a list of regions is returned for each object. Each region and object holds information for contained terms and spatial features.

1.2.1 Pollination

The DWD measures and publishes the air pollination in Germany on a daily basis. These pollen measurements are grouped by natural area.[2] Each region contains additional information, like the concentration or forecast data, which could be given as an output in case of a match, but this is not considered for this example. The retrieved information can be stored as polygons with a key indicating the pollen type as illustrated on the left-hand side of Figure 1.4.

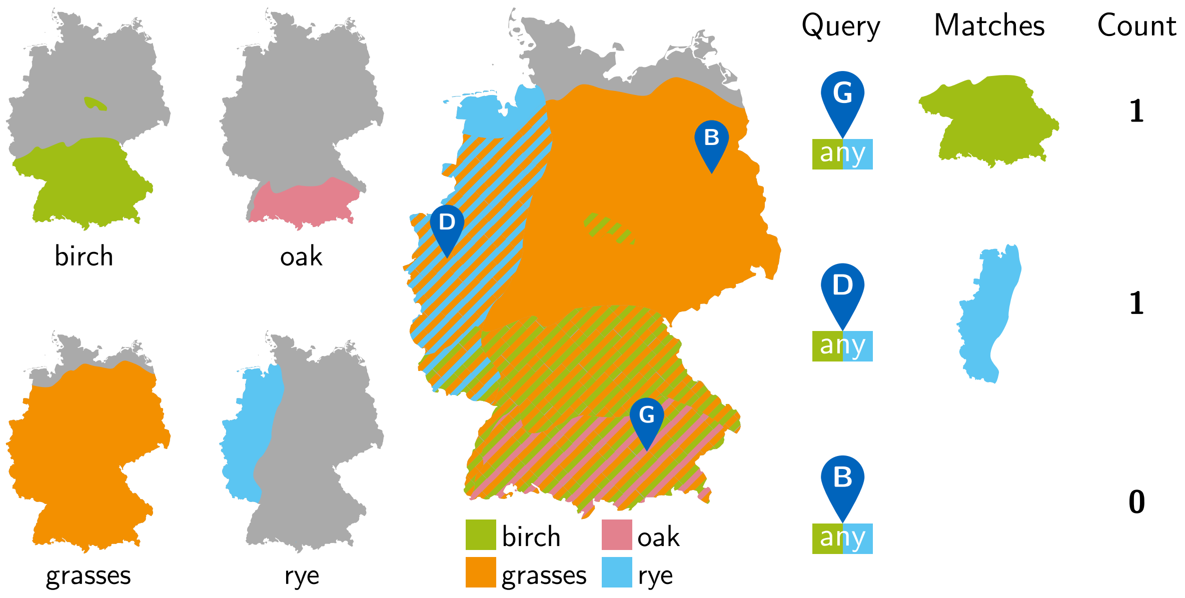


Figure 1.4: Schematic Example of Pollination in Germany

An index combines all polygons which are shown on the left into a data structure containing

[2]The Deutscher Wetterdienst publishes the dataset publicly available at `https://www.dwd.de/DE/leistungen/gefahrenindizespollen/gefahrenindexpollen.html`

the geospatial information along with their respective pollen type. This index now contains all relevant information for getting local information about pollination.

If the user is interested in the pollination at a specific place he has to query the index with the location and his pollen type interest as input. The index has to test whether the user is in the region and whether the requested pollen type matches.

This is a typical query task with smartphones where users retrieve regional information based on their pre-selected settings. Pollen information can be stored on the smartphone and the phone queries the status based on the current position. In case a new forecast is released, the user may be notified.

A geospatial index is not limited to this type of request as shown in Figure 1.4. A user being allergic to birch and rye is studying in Garching. He wants to be at a place where neither pollen is in the air. He has to issue geo-textual joins for each place he is interested in. Each join simultaneously checks all polygons for a match in pollen type. In this example he gets a result for Düsseldorf with one pollen type he is allergic to and Berlin with no pollen type that he wants to avoid present.

In this example, the number of results is shown in the rightmost column. Depending on the index configuration he can get various additional results. In case additional information like concentration is available this may be returned as well to allow a more differentiated result analysis.

1.2.2 Advertisement

The advertiser's prime goal is to personalize advertisements as much as possible. This is especially crucial for local businesses like restaurants or fitness studios which cannot profit from customers outside the city. Hence, the area around the establishment can be modeled as a polygon to spatially limit the advertisement range.

Additionally, big companies like Google, Apple or Facebook analyze massive amounts of data to generate user profiles. Based on them, companies improve advertisement personalization to efficiently address user groups by their interests.

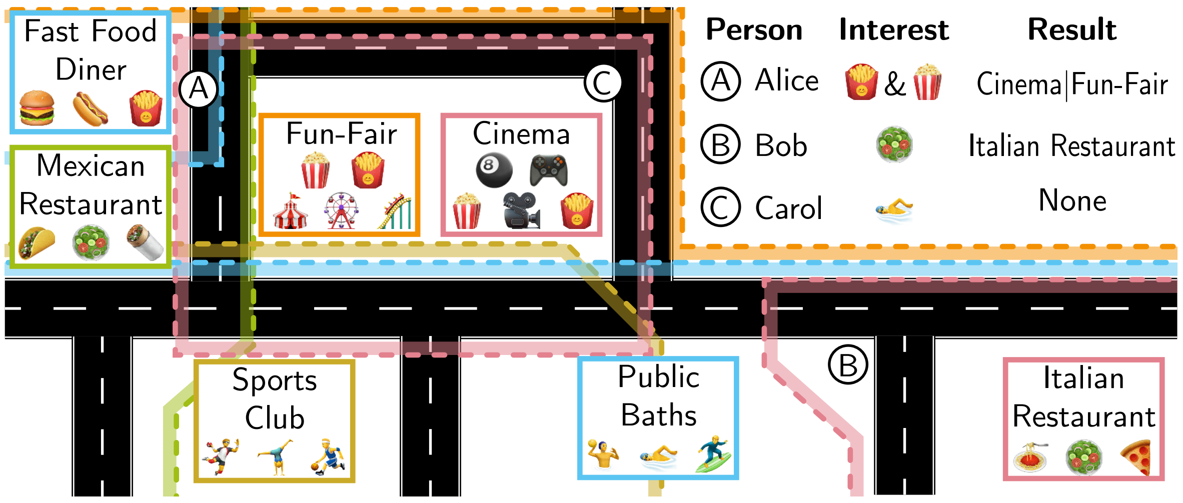


Figure 1.5: Schematic Example of Advertisement

As shown in Figure 1.5 with symbols for different categories, each place has an assigned region which is shown by the dashed lines of the same color, while the inner border marks the inside. Each users' smartphone sends the location to the company coordinating the advertisements. The company itself holds information about the user as shown in the top right corner.

By combining the data, the user gets the best matching advertisement for his profile. This can lead to a win-win situation. On the one hand, companies do not spend their money to show their services or products to people who are not interested in general or not nearby. On the other hand, users get advertisements which might be of interest or even helpful like dinner recommendation while thinking of where to go for supper.

In the example, Alice is looking for french fries and popcorn and standing next to the diner. She is inside the regions of the Mexican restaurant, the diner fun-fair, and cinema but only the last two are offering both french fries and popcorn. Hence she will get a recommendation for one of these places.

Bob is known to like salad. He is getting a recommendation for the Italien restaurant. The Mexican restaurant is not considered because he is far outside the region. Carol is looking for a place to swim. Since only cinema and fun-fair are advertising in her region, she cannot get any tailored advertisement in this scenario.

This workload is notably different from geospatial tasks due to high numbers of spatially overlapping queries. These queries can also be spatially indistinguishable. As a result, spatio-textual indexes must combine spatial and textual features to efficiently process this data.

Datasets with real-world points of interest data are presented in Section 4.

1.3 Problem Definition

This thesis studies the problem of matching a stream of objects O against an arbitrary amount of pre-registered regions R. A spatio-textual data region r contains a polygon which holds the information about the covered region and an alphabetically sorted set of terms. These regions are registered to the index which stores them within the data structure.

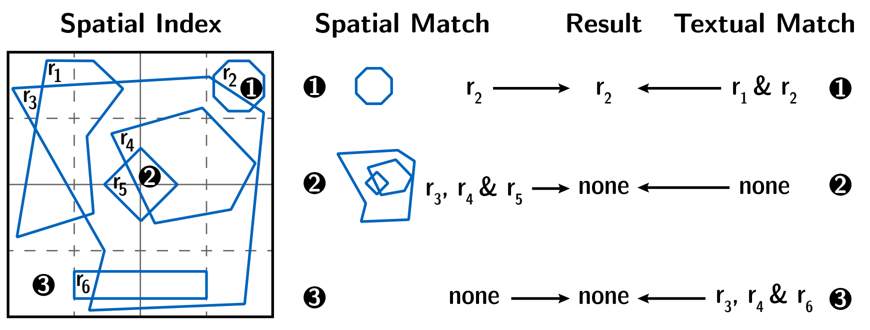
An object o contains the location and a set of terms in alphabetical order. The objects are not known in advance and each object is independent of all others. However, a sample set of objects may be known in advance.

Notation	Descrition
O	A stream of spatio-textual objects
o	A spatio-textual object
R	An indexed set of spatio-textual regions
r	A spatio-textual region
$o.pt$	The geo-location of an object
$r.poly$	The polygon (spatial range) of a region
$r.mbr$	The minimum bounding rectangle of a region
$o.terms \mid r.terms$	The terms of objects respectively regions

Table 1.1: Notations used through the Thesis (inspired by [MAA17])

Figure 1.6 shows how a spatio-textual join is performed for six regions and three objects.

The figure is divided to show the spatial matching on the left and the textual matching on the right side.



	Terms
r_1	Alfa Bravo
r_2	Alfa Bravo
r_3	Charlie Delta
r_4	Alfa Charlie
r_5	Delta Echo Foxtrott
r_6	Alfa Charlie Delta
❶	Alfa Bravo Delta
❷	Bravo Delta Echo
❸	Alfa Charlie Delta

Figure 1.6: Problem Defintion

1.3.1 Geographical Containment

For each geospatial join, it must be tested whether the point is inside the corresponding polygon which can be time-consuming since it may require a point-in-polygon test. For some workloads the precision of the join has to comply only with a certain threshold value. Hence, two types of containment checks are possible for geospatial joins.

Unlike ACT, this thesis uses only exact point-in-polygon matches for all evaluations.

Exact match Whenever $o.pt$ is returned as inside $r.poly$ and all points inside the polygon are returned. In case a used index structure differs between candidate and true hits, every candidate hit has to be tested with a point-in-polygon hit and is either returned as true hit or dropped.

Approximative match The user provides the additional parameter ϵ which indicates the tolerance. Every wrongly returned $o.pt$ has to be in a range of ϵ around the border of $r.poly$.

1.3.2 Textual Matching

Each regions term set has to be tested against the objects term set. In this thesis, the definition for subscription indexes is used which states that all terms of the region have to be present in the object.

$$r.terms \subseteq o.terms$$

It models a user with a wide variety of interests. The user thereby subscribes to information which is fulfilling all his interests.

1.4 State-of-the-Art Methods

A variety of different techniques exists to efficiently store and process spatio-textual data. Usually, these approaches can be divided into three components. A spatial index, a textual index and the combination scheme of both indices. Addressing these components separately leads to a more sophisticated analysis because the weaknesses and strengths of the different parts can be pointed out.

For a general overview of the topic, building blocks of the most recent spatio-textual indexes were considered. These results were combined with the conclusions of a comparison from 2013 [Che+13]. Additionally, some spatial- or textual-only approaches and ideas are presented which may also be considered for use in a more efficient combined index.

1.4.1 Spatial Indexes

This section compares four different spatial indexing schemes. Most of them are primarily used for CPU-centered data processing, but recently GPU-resident indexes were introduced as well.

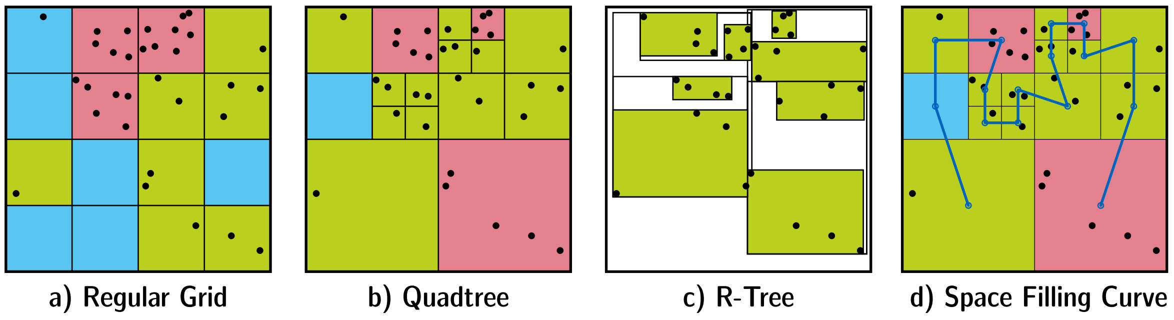


Figure 1.7: Different Types of Spatial Index Structures (overfull or underfull cells are marked red or blue)

Regular Grid Indexes in this category store the data based on x- and y-coordinates in grid cells which all have the same dimensions. This cannot adapt to data locally and tends to lead to many over- or under-full cells as shown in Figure 1.7a. Due to this restriction more flexible grid schemes are used if possible.

When processing spatial data on the GPU, [Zac+17] this index has to be used in order to use the native rasterization pipeline. This is, along with the memory transfer to the GPU, one of the main drawbacks of this method.

Quadtree These indexes partition space in hierarchic quadratic cells of different sizes. Hence x- and y-coordinates are used along with information on the level. Therefore, the cells can be adapted to local data densities. The index itself can be stored as a recursive data structure in a tree or the parameter combination of x, y, and level can be used as a key to index the cells in a hash map.

Using a hash map ensures a linear access time to cells independent of their depth and it is easier to implement. This method is chosen by the authors of FAST [MAA17]. A tree-based implementation preserves the spatial structure and simplifies traversals from root to leaf.

R-Tree The resulting bounding boxes allow a flexible partition of the space. This structure groups nearby points to their minimum bounding rectangles [Gut84]. Different heuristics exist like topographic splitting to minimize the overlap and cell size thus making the process more efficient [Bec+90].

The index is stored in a tree and can be annotated with various additional payloads. Thus most indexes use or built using an R-Tree for data processing. Various approaches use it to improve the pruning power. Li et al. [Li+13] annotate the path with terms, Felipe et al. [FHR08] are using bitmaps.

Space Filling Curve These indexes basically use the same space division as Quadtree indexes but store the data in a completely different manner. The position along the space-filling curve is used to get a distinct number for each cell. Some indexes use this number to effectively sort the terms in cell order. [CSM06; Chr+11] Due to the spatial locality of space-filling curves, nearby data points are stored next to each other. Hence there is also no need to store the cells in a trie or hash map because the curve implicitly orders the points.

Other indexes like ACT [Kip+18] use it along with a trie as a base structure for their index as described in Section 2.1.1.

1.4.2 Textual Subscription Indexes

The used textual indexing schemes greatly differ in complexity depending on the robustness of the index. Indexes for full-text search are not considered in this thesis as the problem is reduced to matching multiple terms.

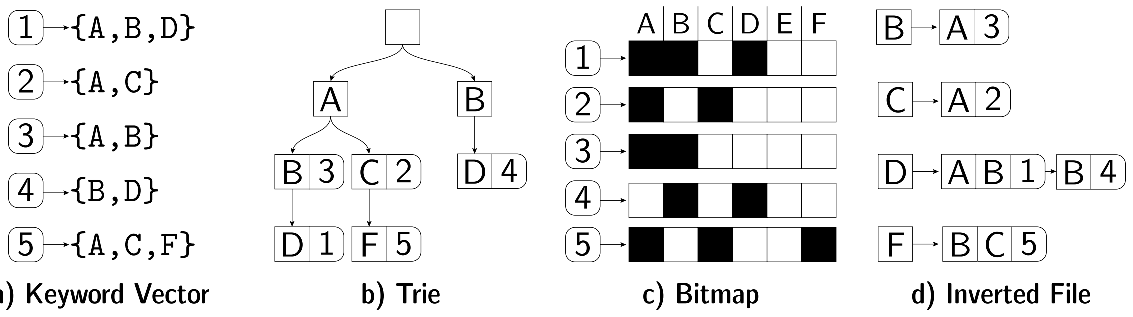


Figure 1.8: Different Types of Textual Index Structures

Vectors of Terms The most straightforward way to index is storing all of the terms in a vector. In case the number of indexed queries is small, using vectors is a fair choice because the overhead is small. It is advisable to store the terms in a sorted order to speed up comparisons. However, when the number of queries is higher, checking every indexed query is not a reasonable option. Thus it is not used as main textual index.

Trie When the terms are evenly distributed and the amount of terms per region is limited, tries are a grounded choice for the index structure. It ensures a high fan-out while an efficient matching of the terms is possible. Techniques like path compression can further be used to speed up the processing. At the time of writing, no spatio-textual structure directly uses them. However, the structure is popular for subscription indexes [Hme+12] and is a building block for efficient spatio-textual structures as shown in Section 2.2.

Bitmaps When the number of distinct terms is low, bitmaps are a reasonable choice for indexing the terms. It reduces all necessary operations on the term sets to binary operations. When the number of terms is too high, the bitmaps are not densely filled. This leads to a significant amount of used memory and slows down the operations. However, some index structures use condensed bitmaps as signatures to allow pre-selection like the IR^2 tree [FHR08] or W-IBR tree [Wu+12].

Inverted File An Inverted File contains a vocabulary of terms and each of them is associated with an Inverted List. Each list contains a sequence of postings, each of these holds all the associated terms and the identifier. Commonly, the lists are sorted by their id. However, some index structures like SKIF [KSL10] or SFC-Quad [Chr+11] order them to benefit from the geospatial structure. In general, Inverted Files are the most widely adopted data structure and used by most of the other indexes like [CSM06; Zho+05; Vai+05].

1.4.3 Combination Methods

There are several different methods to combine spatial and textual pruning. When building based on an existing index, most resulting indexes either consist of a chain of spatial and textual indexing or a loose combination of both. Especially designed spatio-textual indexes often interweave the filtering steps to achieve earlier pruning.

Spatio-First This is the most common choice because efficient spatial indexes without textual discrimination often need to match more features. Almost any spatial index can be used because the textual index can be seen as post-processing for the returned data. Usually it is assumed that spatial features are more selective than textual, so filtering spatial first is a reasonable choice.

Due to the possibility to group nearby regions, most index types can be used as textual part. The number of regions per textural index can be varied easily. Thus the spatial index is a good choice for the superordinate index and is the most widely used combination.

Textual-First Beginning with textual selection leads to a comparable setup as spatial-first indexes. The actual implementation differs because grouping nearby queries in a textual context is not as obvious as doing so with spatial data. The Spatial Inverted Index (S2I) uses R-Trees for indexing frequent terms while aggregating the infrequent items. [Roc+10]

Loosely Combined A better choice when enhancing a data structure is combining information from geospatial and textual data. In contrast to spatio- or textual-first indexes where one part of the index is not aware that another dimension of data exists. Hence for a spatio-first index, the textual part of the data structure might notify the spatial structure when the selectivity of the terms reaches a certain threshold.

These hints can be used to limit the depth of the generated index and avoid unnecessary specialization of one part of the index. The FAST index [MAA17] is using the textual index to initiate splits when necessary and distribute the terms dependent on their frequency.

Tightly Combined Some index structures interweave different filtering steps of text and geographic data. We differentiate between a tight and a loose combination because some indexes use the respective other structure for hints to optimize the building. This communication is referred to in the thesis as a loose combination because technically the indexes can be separated. A tightly combined index can filter either parallel in both dimensions or alternate the filtering.

The AP-Tree [Wan+15] consists of layers for textual and spatial pruning. These are adaptively combined to achieve a more selective filtering. The R^T-Tree [Li+13] stores the terms along the path in an R-Tree to prune textually while refining geographically.

1.5 Structure and Content of Thesis

This thesis is structured roughly in the same order as the parts were developed. Overall, it starts with the explanation of the basic building blocks and how they were combined for developing an efficient data structure in C++. It ends with an evaluation against state-of-the-art competitors and summarizes how the runtime benefits were achieved.

Chapter 2, Building Blocks, lays the foundation for this thesis by presenting the underlying data structures. It focuses on ACT in Section 2.1, which is the foundation of the newly built index and AKI in Section 2.2 which adds the textual pruning power. Section 2.3 describes the approaches taken by the authors of FAST to combine spatial and textual indexing in a sophisticated way. Further ideas to optimize spatio-textual data structures are presented in Section 2.4

Chapter 3, Efficient GeoTextual Joins, describes the actual implementation of the data structure. The implementation of AKI for textual selectivity and several enhancements is presented in Section 3.1 along with the respective benefits. ACT is extended by textual discrimination using filtered lists and AKI nodes in Section 3.2. Alongside this process, the limitations of this approach are sketched.

Chapter 4, Data Generation, presents the used approach constructing three different datasets. At first, a method is described in Section 4.1 that generates regions of different shapes from points. Afterwards, these methods are used in Section 4.2 to generate various uniform datasets. Section 4.3 describes how natural dataset characteristics can be incorporated for more naturally distributed data. Using the publicly available Yelp dataset, Section 4.4 depicts the steps to process the data into a suitable format for spatio-textual joins.

Chapter 5, Evaluation, uses the implementations presented in Chapter 3 and the datasets from Chapter 4 to measure the data structures against their respective competitors. At first, each dataset is presented in Section 5.1 with some key facts. In the following Section 5.3, AKI is compared with its competitors using different hash table implementations. Finally, the spatio-textual indexes are compared in Section 5.4. This section shows the benefits and the scalability of the FAST C++ and the textual ACT implementation.

Chapter 6, Conclusion, ends the thesis with a short retrospection of this thesis' achievements. It also addresses the data structures' limitations and potential future work in this area.

2 Building Blocks

A spatio-textual index needs a data structure which can handle both textual and spatial selection. This is usually achieved by coupling two different structures and combining their pruning power. This chapter schematically explains several building blocks used to build an efficient and performant spatio-textual index.

2.1 Adaptive Cell Trie

The Adaptive Cell Trie (ACT) [Kip+18] is a fast geospatial index for indexing polygons and probing millions of objects per second. It artfully uses main memory to transform the compute-intensive point-in-polygon problem to a memory intensive problem.

2.1.1 S2

ACT employs S2 [Vea+18] for handling all geometric primitives like coordinates, polygons and functions as point-in-polygon tests. The library has a rich set of functions for processing data, both in discrete and non-discrete space.

The main contribution is a mapping from the non-discrete latitude-longitude coordinate representation to a discrete 64-bit value called S2CellId with an accuracy of about 1cm.[1]

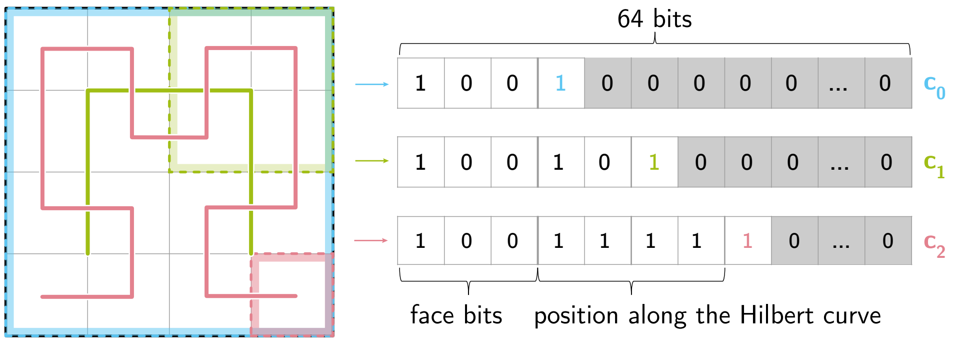


Figure 2.1: Cells at Different Levels Enumerated by S2 using the Hilbert Curve (Figure adapted from [Kip+18])

S2 transforms the globe into a cube where each face of the cube is a two-dimensional plane. On each of these six faces, a space-filling continuous Hilbert curve is used to assign each 2D position a value along the curve. Due to the recursive structure quadrupling the number of cells each step, the position along the curve can be stored as a sequence of bit pairs.

[1]This resembles a maximum error of less than 10^{-7} degrees for latitude or longitude.

Thirty of these bit pairs are stored along with 3 bits determining the face of the cube. The last bit is used as stop bit to indicate the level of the cell. So only the bits left to the rightmost set bit contribute to the position as shown in Figure 2.1.[2]

Using this mapping, all coordinates are simplified to unsigned integers. Due to the properties of space-filling curves, neighborhoods in coordinates are also preserved during the mapping.

2.1.2 Cell-Trie

Using S2, all indexed polygons are converted into two lists of S2CellIds. One list holds the border cells of the polygon while the other contains interior cells. This differentiation helps to avoid time-consuming point-in-polygon checks which are redundant for the interior cells.

Since the S2CellIds preserve the locality to a certain step, they can be used as keys in a trie. The first three bits are handled differently because they indicate the face of the cube. Hence a radix-trie is built for every face with optional path compression at the first level which speeds up the access when the polygons only cover small regions.

The remaining 60 Bits which encode the position along the Hilbert curve were split in chunks of 8 bit each. Thus a byte can directly be used to access the correct node in the trie. The amount of trie-levels depends on the required accuracy and the available memory.

The more cells a polygon is divided up the less of it is indexed in border cells which require an additional containment check. When the cell size is smaller than the required accuracy, containment checks can be omitted which additionally speeds up data processing.

Due to the regularity of the structure and the simplicity of the individual operations, vectorization can be used to additionally speed up the processing.

2.1.3 Payload Marker in Pointer

ACT is designed for 31 bits of payload which are currently used to store the polygon id. The payloads are inlined in the trie's leaves to avoid unnecessary cache misses. To distinguish normal pointers from special values, the value's last two bits are used. It is safe to store additional information there because all pointers are 8 bit aligned which leaves the three least significant bits unset. Hence this 3 bits do not store information.

Depending on their content, the trie-node has one of the following types:

Pointer The pointer is not altered because the tag is zero by default. Thus, it stays correct and can be dereferenced without changes.

Single Payload One 31 Bit value is stored.

Double Payload Two 31 Bit values are stored.

Offset Offset to a lookup table is stored. (Similar to a single payload)

The type of the corresponding node is determined by the number of attached queries. Whenever there are more than two queries, the offset node is used. Thus, the index reaches optimal performance for non-overlapping queries since it avoids using the lookup table.

[2]The whole process to transform a coordinate pair into an S2CellId is explained in detail at `https://s2geometry.io/devguide/s2cell_hierarchy`

2.2 Adaptive Keyword Index

The Adaptive Keyword Index (AKI) is a lightweight subscription index presented as one of the main contributions of the FAST index [MAA17]. It combines the advantages of some of the two most widely adopted textual-only structures Ranked Inverted List [ZM06] and Ordered Keyword Trie [Hme+12].

All index structures are presented and an example for the construction is shown with the dataset below. In this section, spatial discrimination is not considered.

Region	Terms
r_1	t_1, t_2
r_2	t_1, t_2
r_3	t_1, t_2
r_4	t_1, t_3
r_5	t_1, t_2, t_3
r_6	t_1, t_3, t_7
r_7	t_2
r_8	t_3, t_6

Table 2.1: Textual Dataset for Index Examples

2.2.1 Ranked-Key Inverted List

Ranked Inverted Lists (RIL) are a special type of inverted files [ZM06]. Each region is added only to the least frequent term's posting list. This is done using Algorithm 1. Since only one term is stored within the structure of the index, each element of the posting list has to store all remaining terms.

Algorithm 1 RIL Construction

```
1: procedure ADDTORIL($ts$, $v$)            ▷ add subscription $s$ with terms $ts$ and value $v$
2:     $minTerm \leftarrow$ GETLEASTFREQUENTTERM($ts$)
3:     $postingLists[minTerm]$.APPEND($\{ts, v\}$)            ▷ add to the posting list
```

To determine the least frequent keyword, prior knowledge of all terms is necessary. This limits the practical usability as the total dataset is not necessarily available. To overcome this problem a training dataset may be used. Alternatively, only the currently indexed terms can be used to approximate the real data, called adaptive RIL throughout this thesis.

A RIL balances the length of the lists because the posting lists of infrequent terms are chosen for insertion which tend to be short. Furthermore, the structure does not have much overhead which results in a small memory footprint. The worst case for inverted lists is a high number of regions with a small set of terms. This forces the regions to be indexed at frequent terms leading to long posting lists.

When matching an object, the corresponding posting list is retrieved for each of the objects terms. Every element in the list has to be checked whether it contains a subset of the objects terms. This procedure is shown in Algorithm 2.

Figure 2.2 shows how the dataset, presented in Table 2.1, is inserted into an Adaptive RIL. The regions r_1 to r_8 are added sequentially and every node which is generated along the insert

Algorithm 2 RIL Matching

```
1: procedure MATCHRIL($ts$)                          ▷ get values corresponding to terms $ts$
2:     $result \leftarrow \{\}$
3:     for each $term \in ts$ do                     ▷ get indexed elements for every term
4:         for each $element \in postingLists[term]$ do           ▷ test all elements
5:             if $element.terms \subseteq ts$ then
6:                 $result \leftarrow result \cup \{element\}$    ▷ add match to the result set
7:     return $result$
```

is marked in the same color as the inserted region itself. Each node is added to the term which has the lowest frequency at the time of matching. In case of a tie, the region is inserted to the shorter posting list. If this did not resolve the tie, a random node gets the element. In our example the random node is always the first in alphabetical order. Using these rules a nicely balanced RIL is obtained. The insertion process is shown in the left two subfigures of Figure 2.2.

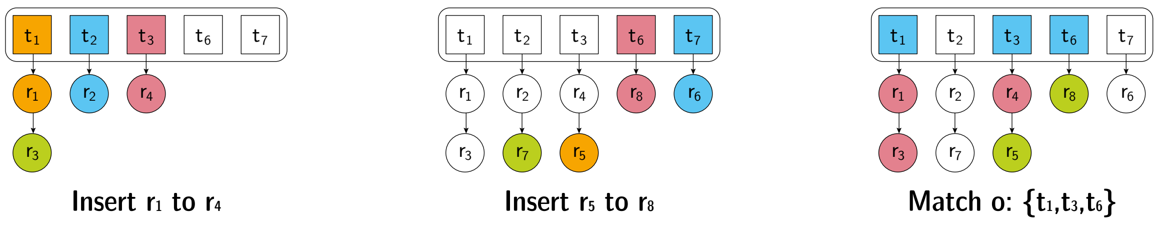


Figure 2.2: Adaptive Ranked Inverted List Insert and Match

The rightmost part of Figure 2.2 shows how an object is matched. At first, the algorithm visits all list heads marked in blue because the term is in the objects terms list. Afterwards, all lists are traversed and checked whether the terms fully match. For example all nodes of the t_1 list are not returned because they just contain t_1 and t_2 and thus are no proper subset of the queried object. All matched regions are marked in green.

2.2.2 Ordered Keyword Trie

An Ordered Keyword Trie (OKT) [Hme+12] is an alternative to the flat RIL index. It builds a hierarchic trie of sorted terms. Unlike normal trie structures which index strings character by character, OKT indexes subscriptions term by term. Each trie-node represents a term and contains a list of subscriptions. A subscription is added to the list if all terms are included in the path from the root to the node.

Hence subscriptions with similar terms are likely to be stored in neighboring nodes to take advantage of a common prefix. The hierarchical search space speeds up the search time because list traversal like for RIL is replaced by hash table accesses which can be performed in constant time on average.

In order to maintain a valid trie structure, the insertion starts with the first term in alphabetic order. Therefore the nodes of the path have to be totally ordered. If possible, path compression can be applied to parts of the path. This ensures a faster tree traversal and reduces the number of hash table accesses.

Due to the high fanout, more memory space than for an RIL is necessary. However, nearly all accesses are handled by a hash table because the attached lists contain solely true hits. There is no need for comparing and storing the keywords of the indexed regions, unlike the RIL.

Algorithm 3 OKT Construction

```
1: procedure addToOKT($ts$, $v$)        ▷ add subscription $s$ with sorted terms $ts$ and value $v$
2:     $node \leftarrow$ root
3:     while $ts \neq \emptyset$ do                                   ▷ traverse trie
4:         if $node$.hasTerm($ts.first$) = false then
5:             $node$.attach($ts.first$)
6:         $node \leftarrow node[ts.first]$                  ▷ step down in next level
7:         $ts \leftarrow ts.tail$
8:     $node$.appendRegion($v$)                               ▷ add to the node
```

Algorithm 3 shows the insertion of a region with a pre-sorted set of terms. Following the terms, the path through the trie is taken or new nodes are generated. When the list of remaining terms is empty, the subscription is added to the current node.

Beginning with the list of alphabetically sorted terms, each proper subset of the search terms could lead to matching regions. This can be achieved elegantly by applying recursion as shown in Algorithm 4. All regions attached to the traversed nodes are true hits and can be directly added to the result set.

Algorithm 4 OKT Matching

```
1: procedure matchRIL($ts$, $node$ = root)       ▷ get values corresponding to terms $ts$
2:     $result \leftarrow \{\}$
3:     while $ts \neq \emptyset$ do          ▷ stop loop and recursion when all terms are traversed
4:         $ts \leftarrow ts.tail$
5:         $result \leftarrow result \cup$ matchRIL($ts$, $node[ts.first]$)    ▷ recursively add other regions
6:     return $result \cup node.regions$                    ▷ add current nodes regions
```

Figure 2.3 depicts the generation of an OKT. All inner nodes which are added during the insertion are colored to match the region. The terms are added in alphabetic order to obtain a predictable outcome. All textually equal regions, r_1 to r_3, are inserted to the same node. The full trie has three levels because the maximum number of terms is also three. All rectangular nodes represent the trie structure.

The path to the region contains all region's terms. The attached lists are textually indistinguishable, thus they can neither be shortened nor reorganized.

When matching the object, all terms are looked up in alphabetic order. Thus, first t_1 is tested, then t_3 and so on. In case the leaves are reached, all contained queries match the terms. In contradiction to the RIL, no additional check is necessary. All visited inner nodes are marked in blue and all returned regions in green.

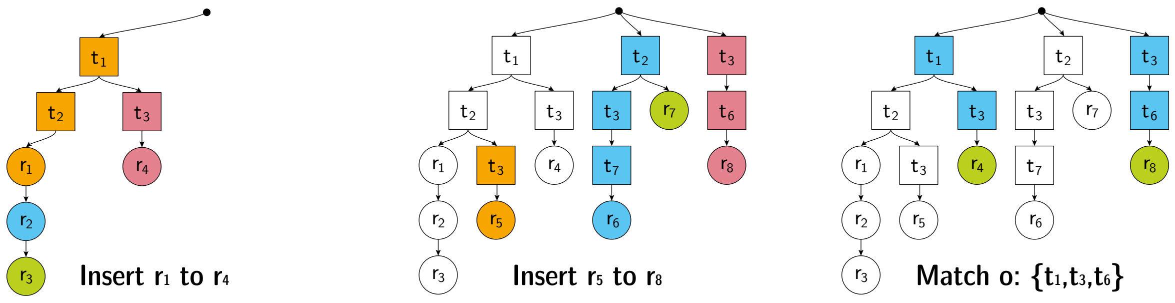


Figure 2.3: Ordered Keyword Trie Insert and Match

2.2.3 Combination

RIL and OKT have different strengths and weaknesses depending on the indexed workload or the used metric. Hence their beneficial features are combined into a more versatile index. The Adaptive Keyword Index (AKI) differentiates between frequent and infrequent terms and uses a fitting data structure. Basically, AKI indexes all frequent regions in an OKT while adding the rest of the regions in RIL manner to the structure.

In the beginning, all nodes are infrequent and contain a list of attached subscriptions. The data structure uses internal counters to keep track of term occurrences. The user can set a threshold θ to indicate when a list is too long and the associated term is considered to be frequent. Until no node is marked frequent, insertion to AKI works the same as to an adaptive RIL which is presented in Figure 2.2.

Figure 2.4 compares RIL with AKI having a frequent threshold θ of 2. That means all RIL lists longer than two will be inserted in an OKT structure. In RIL, r_9 would be simply added to the list of t_3, resulting in a list length of 3. For AKI this results in a potential remodeling. When a nodes posting list would exceed the frequent threshold, AKI tries to reinsert some of the attached regions to other non-frequent posting lists. If this operation is not successful the index has to be restructured.

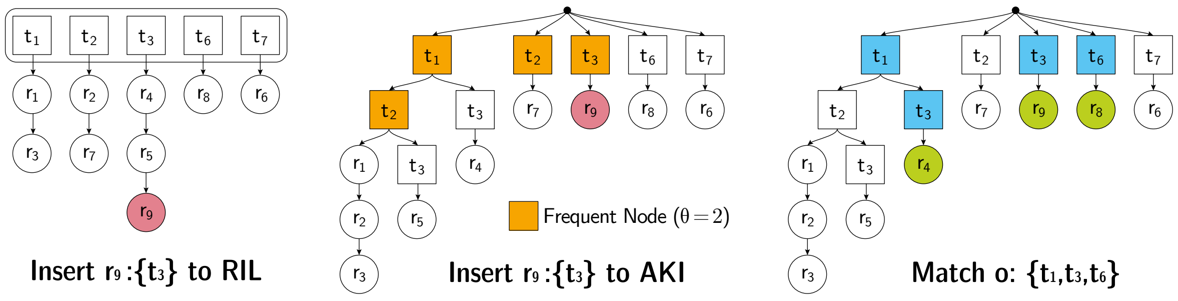


Figure 2.4: Adaptive Keyword Index Construction and Match

All terms of the nodes indexed at t_3 are marked as frequent terms. This additionally makes t_1 and t_2 frequent terms because their items also cannot be redistributed. Hence all regions indexed below these terms are inserted in an OKT manner, explaining the similarity to Figure 2.3. Technically all regions are temporarily stored in a queue and inserted subsequently.

In contrast to OKT, the child nodes are not frequent by default. Each frequent node's children start as infrequent. They are upgraded following the same rules. So both t_3 nodes being descendants of t_1 are not frequent since their attached list is shorter than θ.

Matching AKI is, like the structure itself, a mixture of both matching algorithms. The trie is traversed using the same algorithm as OKT. In case an infrequent node is visited, the match is performed using the same technique as RIL. Hence lists have to be traversed to check whether the terms match but their length is limited to θ.

In conclusion, AKI combines advantages of RIL and OKT to handle frequent and infrequent terms differently achieving good performance for different dataset sizes as shown in Section 5.3. It uses the RIL which provides none to small overhead for handling infrequent regions while indexing all other terms in a trie, consuming more memory. This provides a trade-off between memory consumption and performance as stated by Mahmood et al. [Mah18; MAA17].

2.3 Frequency-Aware Spatio-Textual Indexing

The authors of the FAST index embedded AKI loosely in a geospatial grid index to generate an adaptive spatio-textual index. They used several approaches to communicate between the data structures in order to reduce matching time and memory consumption.

2.3.1 Combination of Spatial and Textual Properties

FAST hierarchically divides the space into cells. Each cell has an AKI instance attached as depicted in Figure 2.5. The refinement of the spatial grid is controlled by the attached textual indexes which try to minimize the number of frequent nodes.

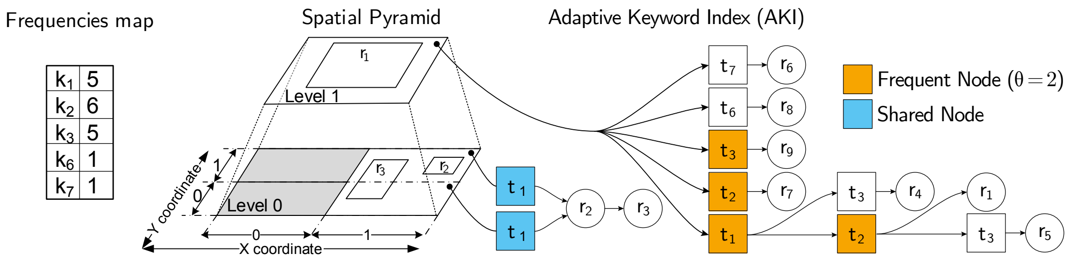


Figure 2.5: Schematic Structure of FAST (Figure adapted from [MAA17])

A reorganization is triggered when the number of textually indistinguishable regions for a specific term reaches a threshold in an AKI instance. The index returns the remaining non-inserted regions and the rest of the list with the command, to reinsert them at a lower level. The control structure generates a new index cell and inserts the subscriptions in the corresponding cell. All other non-frequent subscriptions stay in AKI on the layer above. This is illustrated in Figure 2.5 where the regions indexed at t_1 are spread across cells at different levels.

Therefore, a spatial pyramid is built which indexes the subscriptions on various layers. When matching an object the cells at different granularities have to be tested to find all potentially matching items.

The textual AKI structure is also aware in which cell and level it is located. Before the superior spatial index is notified for refinement, it checks dependent on the cell boundaries whether it is worth the effort. For example, cells at the lowest level cannot be refined spatially anymore.

2.3.2 Shared Frequency Information

A FAST instance consists of cells of which each has its own entity of an AKI attached. Normally, each structure has their own frequency information. Although, the small sample size of each individual index introduces high variances to the frequency.

By splitting the frequency information into local and global parts, all data structures can contribute to the global frequency while still locally having information to decide which term appears to often to be infrequent.

The global information is used to distribute the terms on the various infrequent nodes throughout the tree. Because the sample size is much bigger, the actual distribution is approximated better and the subscriptions are inserted more balanced. In case of a tie or a redistribution of indexed nodes, the local frequency is also taken into account. This ensures local biases are also respected.

Memory consumption is nearly not affected because the local indexes have to be kept for the enlisted purposes. Furthermore, one additional index carries almost no weight.

2.3.3 Spatial Sharing

FAST indexes rectangular regions which might extend over several cells. Typically each cell has its own AKI instance which is independent of all neighboring indexes. This may lead to severe memory overhead at lower levels since the region information has to be duplicated for each cell.

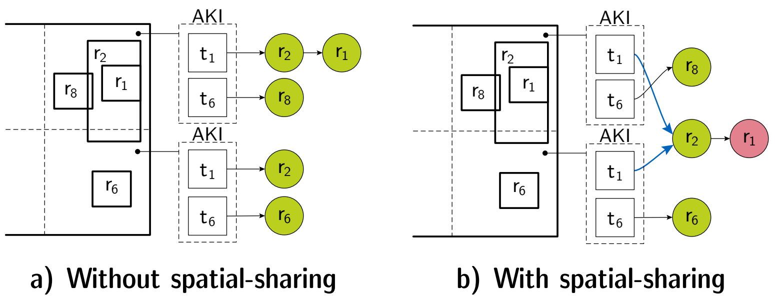


Figure 2.6: Spatial Sharing of Region Lists in FAST (Figure adapted from [MAA17])

Thus, when adding a new region all contained cells are computed and combined if possible to add the region only once. For that reason, it is only added to the top-left AKI node. In case the node is not marked frequent it is possible to share it across several cells on a per keyword basis as shown in Figure 2.6. All other cells simply add a reference inside their AKI to indicate a shared access to the specific term list.

When adding more subscriptions, the shared lists must be handled with care. In case the shared list's number of items stays below the threshold, the new subscription can be added. They may not fall into the spatial range of the cell but since the lists of infrequent nodes have to be traversed in order to obtain only the matching regions, they are pruned during this step.

In case the new region would upgrade it to a frequent node the shared items have to be replicated and attached to non-shared lists for all AKI instances. Now the new item can be added securely because all connections to other nodes regarding this term are resolved.

2.4 Various Methods

The preceding components were categorized according to their associated data structure. Since the inspirations for the index structure are not limited to this sources, some general methods and techniques are presented in this section.

2.4.1 Keyword Dictionary

Spatio-textual indexes contain a high number of terms which tend to have a much smaller vocabulary size since mostly terms or tags are indexed. Because handling character strings, in general, might cause performance problems due to the variable length, the slow comparisons or the comparably large memory consumption, it is a reasonable choice to avoid using them.

The problems can be reduced by a keyword dictionary which maps each individual string to a number. Thus, the problem is reduced to index fixed-length, fast comparable integers along with spatial information.

In order to profit as much as possible from the keyword dictionary, one global dictionary has to be used. Hence directly when adding the region to the index structure, the terms are mapped to the respective numbers. Otherwise, several local directories would exist, slowing down the code by filling up the caches and consuming more memory.

2.4.2 Bloom Filter

Bloom filters, devised by B. Bloom in 1970 [Blo70], are a space-efficient probabilistic data structure to check for set containment. They are typically employed to avoid unnecessary disk or network accesses by providing a good estimate of whether the element exists in the set.

In case the filter rejects the element, it cannot be in the set. Otherwise, it may be contained but chances for false positives exist. The probability of false positives is affected by the relation between filter size and the number of indexed terms. Intuitively the more bits are available for each term the lower is the false positive rate.

When checking for a single term, the term is converted to set bits using several hash functions and checked for containment in the filter. Similarly, for multi-set containment, the set bits of each term can be combined into a single bit-set. In case all the set bits are in the filter, the whole set may be contained. Otherwise, it cannot be contained.

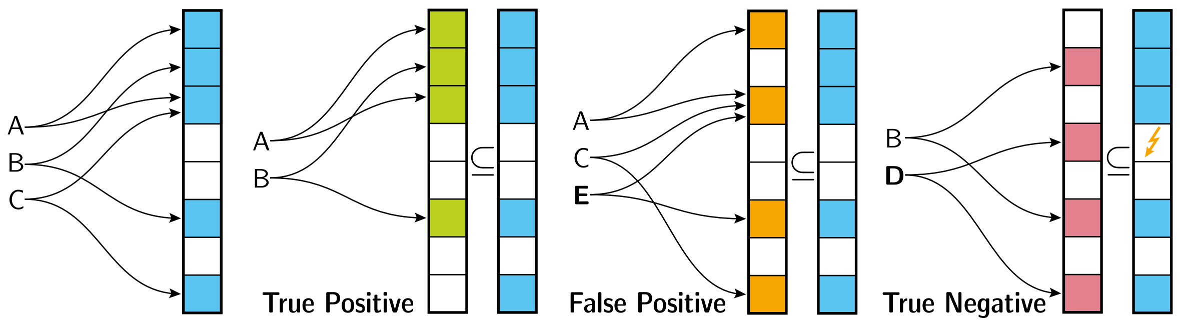


Figure 2.7: Bloom Filters for Multi-Key Containment

This is illustrated in Figure 2.7 for all three possible outcomes. For both true and false positive all set bits are a subset, while this does not hold for the original word set in case of the false positive. The true negative is rejected because one set bit is missing as illustrated in the figure. Due to the design of Bloom filters, false negatives cannot happen.

Thus these types of filters can be used to avoid exact checking keyword set of region and object in some situations. Each indexed regions has a set of keywords. All of them are encoded into the Bloom filter and it is stored along with the keywords.

When checking whether the region's terms are contained in the object, the bloom filter is checked first. Since the keywords do not match in most of the cases, the slightly more costly check of the terms can be omitted. In case the filter returns a possible containment, the term lists have to be checked to ensure it is a true hit.

Since the Bloom filters meaning is reduced to the use as bit-mask, other modern implementations of filters like a cuckoo or variable width filter are not applicable. They do not provide the possibility to combine multiple filters by bit unions or intersections since the set bit position is not solely determined by the hash value. Thus the cost of evaluating the filter is too expensive in comparison to the term containment check.

2.4.3 Hashing Algorithms

Most of the index structures performance is dependent on the used hash table implementation. The problems workload is very lookup centered. Deletions on the index are not considered. Insertions are only performed during the build phase, so their performance is not critical. The whole index is not altered while matching, meaning lookups should be executable in parallel.

Most hash tables, also the gcc standard library implementation, use hashing with chaining. This introduces memory allocations of smaller portions and internal redirections. While this simplifies and speeds up inserts and deletes, lookups tend to involve some redirection steps.

Generally speaking open-addressed hashing, for example with linear probing, is a reasonable choice since the whole table will have no indirections. While this is beneficial for lookups, insertion may have to grow the hash table. This time-consuming step may involve all elements but since it cannot apply on lookups, it does not affect the index matching performance.

Since open-addressed hashing does not assign each element directly to the respective spot, no assumptions can be made for the distance between the calculated spot and the actual spot for the entry to store. This leads to a possible worst-case behavior for non-contained elements. In a trivial implementation, a miss in a full table could result in a full table scan to ensure it is not contained. The following two approaches present a solution to control the maximum and average distance of the actual insertion spot to the proposed spot.

Robin Hood Hashing As the name Robin Hood suggests, this approach takes from the rich and gives to the poor elements. An element is poorer, the higher the distance from its original spot is. When inserting a new element it is checked whether the newly inserted element would profit more from this particular location. In this case, the elements are swapped and it is continued until each element found a place. When the maximum distance between the insertion place and storage place reaches a certain threshold, the table is expanded.

Using this method means that the average distance is not reduced, but the variance of the distance is greatly reduced. It ensures that checking the hash table has a smaller constant lookup time. Furthermore, the threshold effectively limits the number of elements which have to be traversed before safely returning a key as not contained.

Hopscotch Hashing Hopscotch hashing has a similar but more radical approach. It assumes that testing elements in the neighborhood takes the same time as checking the element at the spot itself. Hence it does not matter where the element is located exactly as long as it is near. This property is typical because data is retrieved in cache-lines. So when having one item of a cache-line retrieved, all adjacent items in the same cache-line are loaded simultaneously.

Each bucket has an index to store the positions of its elements for retrieval. In case the index has no more free places to store the elements positions, the hash tables size is increased.

Implementation The implementation of different hash tables is not the scope of this thesis. Hence a comparison of hash table implementations is used to select suitable libraries. These implementations are tested against each other. This work is described in Section 5.3.1.

3 Efficient Spatio-Textual Joins

The previous chapter described building blocks being used to design and implement a data structure capable of handling spatio-textual joints efficiently. This chapter shows how they are combined to obtain an efficient data structure. C++17 is chosen as implementation language because it offers the capabilities to work directly on the hardware, unlike Java which was chosen by the authors of FAST. Furthermore, C++ is used as the implementation language for ACT.

3.1 Adaptive Keyword Index

The Adaptive Keyword Index is deeply integrated into FAST. It cannot be removed in one part in the Java nor in the C++ variant of FAST. Because the different multi-keyword indexes have to be tested, AKI was implemented as a stand-alone textual index structure.

It features a fairly simple interface. One can add new subscriptions and retrieve either a list of subscriptions or their count. This interface is used to implement several subscription indexes like OKT, RIL and some contestants using vectors to simply store the indexed terms.

This section provides some insights into the implementation and optimization of AKI. The resulting data structure is compared with a plain AKI implementation throughout the section to show the speed up. An evaluation against different contestants is presented in Section 5.3.

3.1.1 Node Type Reduction

In Java, the authors of FAST used dynamic casting to a high degree. Each node type could either be a single subscription, a list of subscription or a trie of nodes. In order to store these types, the fact that everything is a descendant of object class was used. Each time the node is being accessed the actual type has to be inferred in order to process the node.

Since C++ does not allow this type generalization to objects and it would affect the performance a more simple way of differentiation was chosen. Each node consists of a list for infrequent queries and an unordered map for frequent queries. A flag indicates whether the node is frequent or not.

The introduced overhead is not big because the list is in use for frequent nodes as well. All textually indistinguishable nodes are stored in the list.

Moreover, Java features a garbage collector cleaning up unreferenced items. As C++ does not have these features, unique pointers are used to keep track of the contained nodes. Because the generated structure is a tree, dangling pointers cannot occur.

In conclusion, our implementation avoids dynamic casting and even casting in general. Memory handling is done implicitly by using unique pointers for ownership.

3.1.2 Keyword Dictionary

The original AKI implementation uses character strings of variable length throughout the whole data structure. All node's keys are strings, every region has a set of it attached which have to be checked for containment.

As long as a string is comparably short, space and runtime overhead is noticeable but not severe. Though when a user begins indexing long strings, like whole articles, the performance would drop drastically.

In order to avoid or constrain the impacts of string handling a global keyword dictionary, as explained in Section 2.4.1, is implemented. The first step in processing an incoming object is converting the strings to numbers.

Hence the data structure is directly working with numbers and the compiler can perform beneficial optimizations. Sorting numbers also works faster than strings and is well defined since they are totally ordered.

To confirm the theory, measurements are performed comparing the plain AKI implementation with the keyword dictionary variant. All measurements are performed with the term data of the Yelp dataset, whose generation is shown in Section 4.4.3. It features a small amount of comparably short strings.

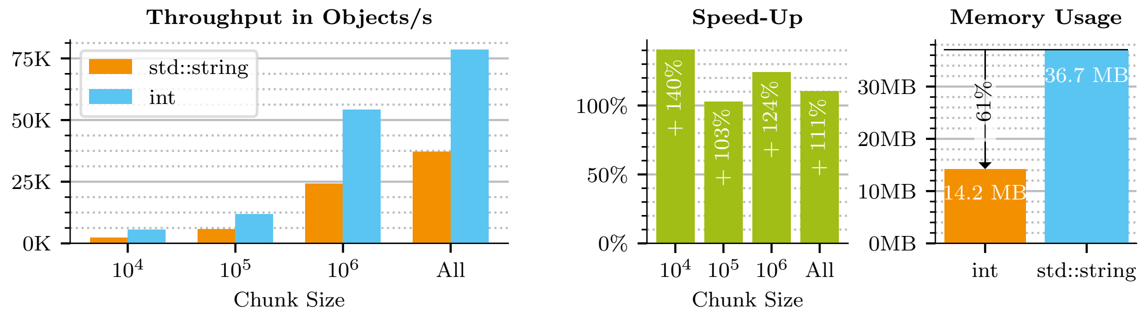


Figure 3.1: Performance Measurements for AKI with and without Keyword Dictionary

Although the strings are short, the memory usage is significantly reduced by 61% as shown in Figure 3.1. The keyword dictionary itself is comparably small and does only take 81KB of memory which is 5‰ of the total used memory.

All measurements use a total of about $170,000$ indexed subscriptions which are split into chunks of different sizes. For this reason, a varying number of AKI instances was generated to simulate the usage in the leaves of a geospatial data structure. Looking at the figure, independent of the chunk size, the keyword dictionary approximately doubles the performance.

In conclusion, a keyword dictionary does not only provide theoretical advantages. It improves both memory usage and throughput. Furthermore, it simplifies writing code since special values for empty or non-existent terms can be defined easily.

3.1.3 Bloom Filter

AKI differs between frequent and infrequent nodes. While subscriptions attached to frequent nodes are directly returned because their terms are encoded in the path, all infrequent subscriptions have to be checked. All its terms have to be contained in the objects terms.

Although only 6‰ of all returned hits come from infrequent nodes, they contain about 2 million subscriptions for this test set. Hence the operation is still very common and has to be performed at least 2 million times for the true positives but in reality, it is performed almost three times more as shown in the first column of Figure 3.2.

A Bloom filter can be useful, as explained in Section 2.4.2, as inexpensive preselection since it only needs simple logical operations. In case it states that the terms cannot be contained, the slightly more costly function call and vector containment check is avoided.

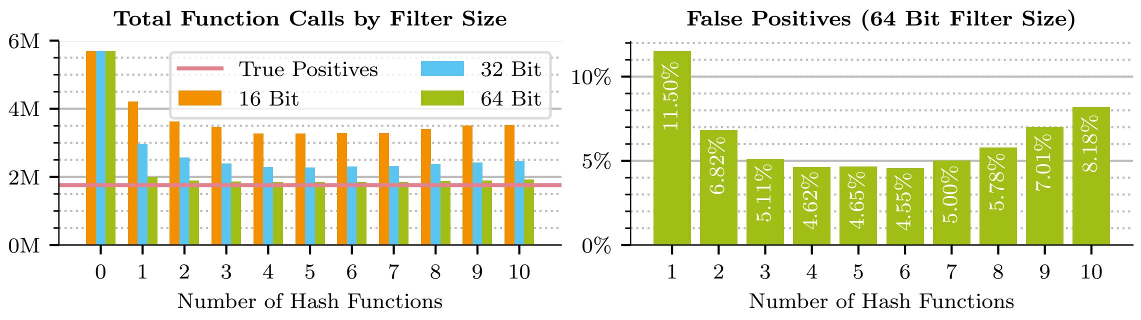


Figure 3.2: Total Function Calls using Bloom Filter

The filter was implemented in three different sizes, to compare the impact of the contained entropy. Figure 3.2 also shows the true positives serving as a lower boundary for minimum required function calls. Obviously, without any hash rounds, the size of the filter does not matter. The smaller size filters were generated by first creating a 64-bit filter with the specified number of hash rounds and than only the respective number of bytes is used.

All filters reduce the number of unnecessary function calls drastically. Without the filter, 68.4% of all calls return false and thus can be omitted. Furthermore, all filters perform best with 4 to 6 hash functions used meaning the number of bits set per term. When the number is lower not enough bits are set for enough entropy, when the number is too high, too many bits are set causing collisions and thus also reducing the entropy.

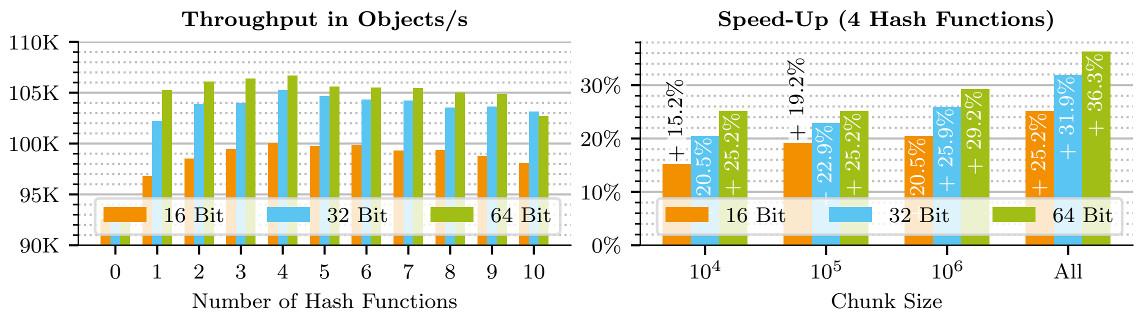


Figure 3.3: Performance Improvements using Bloom Filter

Both 32- and 64-bit filter achieve good filtering results. With a 64-bit filter and 6 hash functions, only 4.55% of the calls are false positives, while using 32-bit filter and 5 hash functions, 22.66% of the calls are not needed. This reduces the number of unnecessary function evaluations by 97.9% respectively 86.9%.

In the actual implementation, 64-bit filter size and 4 set bits were chosen because it requires only 24 bits of entropy. This can be, in contrast to 6 bits, generated using a single 32-bit hash function. To confirm these assumptions further performance measurements were performed as shown in Figure 3.3.

The throughput is increased for all three filter sizes, but the highest throughput is measured when setting 4 bits per term, as theoretically stated above. This leads to a speed-up of up to 36% in comparison to an unaltered AKI implementation.

These results do also match the theoretical backgrounds of bloom filters. m is the number of required bits, which is fixed to 64 bit and n is the number of elements. According to Table 5.2, the Yelp dataset has an average of 8.8 terms per object.

$$k = \frac{m}{n} \ln 2 \Rightarrow \frac{64}{8.8} \ln 2 \approx 5$$

According to the formula, using five hash functions results in an optimal number of set bits, which matches Figure 3.2 since it is approximately symmetric around five. The measured optimal value of 4 is close to the theory. In the actual implementation, the number of hash functions is calculated dynamically using the formula above based on a sample of the dataset.

The theoretical false positive probability is given by the following equation.

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p \Rightarrow p = 2^{-\frac{m}{1.44 \cdot n}}$$

This results in an optimal false positive probability of about 3% for a single lookup. The average false positive rate with l independent lookups is calculated as follows.

$$p_l = 1 - (1 - p)^l$$

Using $l = 3.8$ as average region's term number from Table 5.2 for the Yelp dataset, 10% false positive are estimated. This is higher than 4.6% which are measured as shown in Figure 3.11. Thus the actual lookups are not independent.

The false positive rate increases with the number of lookups and finally reaches 50% when more than 22 terms are probed. Because the datasets do not feature terms lists of this length, using the filters provides benefits for every used workload.

In conclusion, bloom filters perform better in the actual implementation in theory and their performance limitations are not likely to be reached for realistic datasets.

3.1.4 Term Reduction

As stated before, AKI handles infrequent and frequent items differently. When indexing a frequent subscription, the item is stored in an OKT inspired way. OKT is not storing terms attached to the item, as described in Section 2.2.2. All relevant terms are encoded in the path to the nodes.

AKI is not using a fully branched tree. Hence some leaves still have short lists of infrequent queries attached. Though there is no necessity to store all terms along with the item because at least some have to be contained in the path. Otherwise, it would not be a frequent node or the algorithm would not have reached the node.

Thus when transferring items from the infrequent to the frequent part of the data structure, all terms which are contained in the path of the node are marked obsolete and deleted subsequently. As a consequence, the number of keywords is reduced by the number of the node's level in the trie. All elements in the textually indistinguishable list of the node have no terms attached to their list.

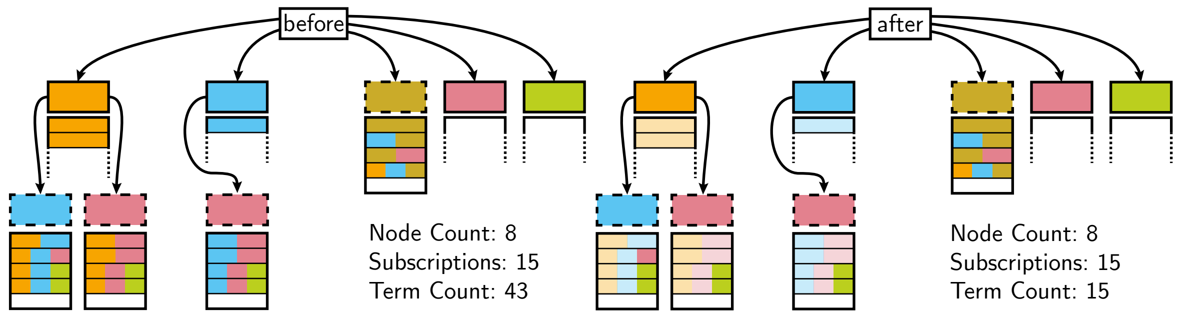


Figure 3.4: Reducing Terms in AKI

This process is illustrated in Figure 3.4. On the left-hand side, the starting situation and the total number of terms are shown. On the right-hand side, the reduced tree is shown. Each color stands for one indexed term. Every infrequent node is shown with a dashed border. Since only the frequent nodes and its descendants are considered, the brown node remains unchanged.

For all other nodes, every term which appeared during the path to the node is deleted. This is symbolized by fading out the terms. The subscriptions directly indexed under frequent nodes, like blue and orange, need no terms attached.

For all other descendant nodes, the number of terms is reduced by the number of terms in the path. Although Figure 3.4 just gives a simple example, the number of terms is reduced by two thirds.

As a consequence, the average number of the attached terms when performing subset containment operations is notably reduced as shown in Figure 3.5. The reduction is greater the larger the number of indexed subscriptions is. When indexing more subscriptions, more nodes get frequent and the trie will nest deeper. Hence there exist longer paths where the terms can be reduced.

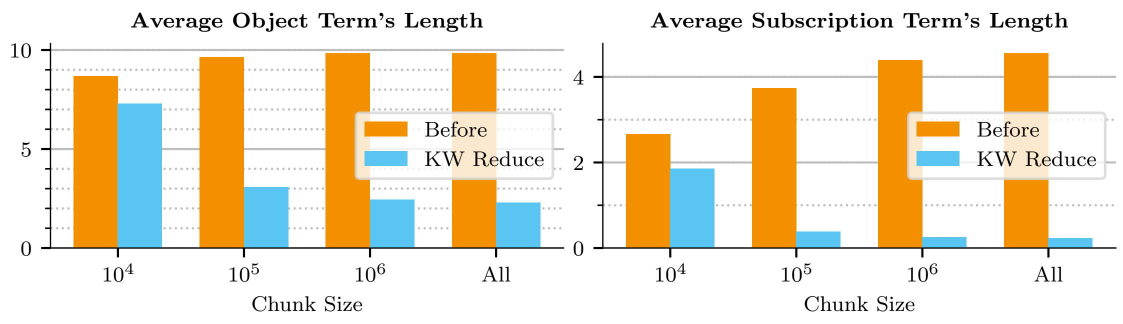


Figure 3.5: Average Length of Term Lists

Using this method, the majority of all terms can be deleted during the insertion as shown in Figure 3.6. Additionally, the terms which are matched are also consumed level by level. For this reason, the number of terms reduces for each level by one both for the object which is searched and the indexed query. Thus a significant amount of terms can also be skipped when querying AKI.

The average subscription length tends to get smaller than one. This case is handled separately because empty lists do not need a containment check. As a consequence, the number of function calls is reduced significantly as shown in the rightmost subfigure.

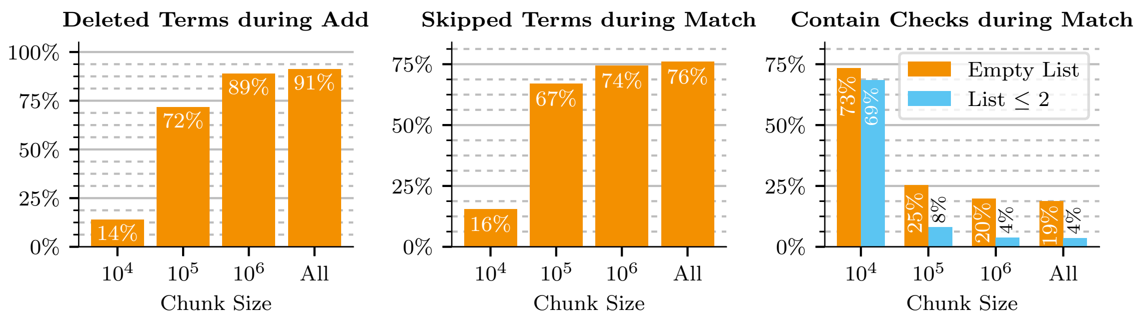


Figure 3.6: Benefits of Optimized Term Processing

The corner case of a one element list can also be handled differently because only one of both term lists has to be traversed. Figure 3.7 shows the performance benefits using the presented keyword reduction technique.

All variations lead to a speed-up of more than 10%. Some additional acceleration can be achieved using different handling for the special cases. To reduce the cost of the function call, it was inlined.

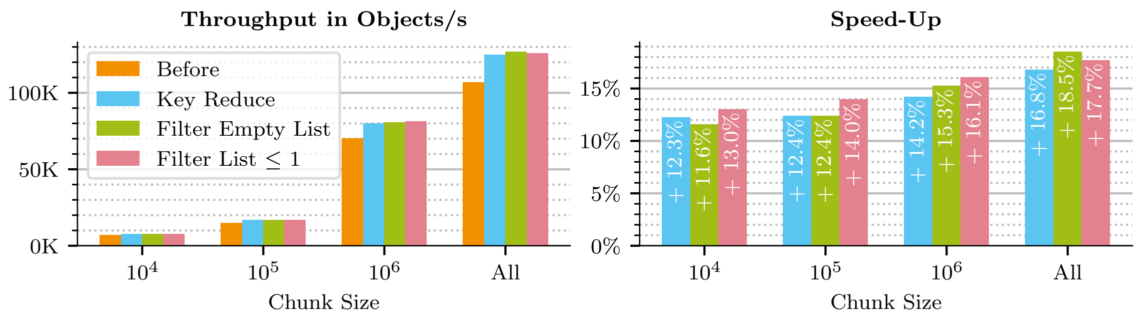


Figure 3.7: Performance Increase due to Term Reduction

The bloom filters attached to the nodes are not altered when reducing the number of terms. Since all indexed terms have to be contained in the object, reducing the number of indexed elements would actually mean reducing the bloom filters selectivity.

Altogether the AKI implementation is massively optimized using a keyword dictionary, small bloom filters and reducing the terms in the structure. As a consequence, the matching throughput more than triples. Further evaluations with other hash table implementations and comparisons with other subscriptions with multiple datasets can be found in Section 5.3.

3.2 Extend ACT by Textual Discrimination

ACT is a geospatial index optimized for modern hardware. It cannot differentiate between regions with the same spatial extent differing textually, as no textual support exists. This section explains how textual discrimination can be added, using some of the building blocks described in Chapter 2 and the optimized AKI implementation described in the previous section.

The support is added in several steps from a working prototype for correctness check to an efficient implementation. During each iteration, correctness is ensured by tests. The uniform dataset with rectangular extent described in Section 4.2 is used because it fills the entire domain. In some cases where only a textual matching is performed, the Yelp dataset known from the previous section is used additionally.

3.2.1 Prototype for Correctness

The prototype leaves ACT structure unchanged and provides a strictly divided spatial-first index. It stores all term data in a separate data structure indexed by the associated region. ACT returns all nodes assigned to the queried cell. These are divided into internal and border cells. Only border cells need an additional verification step to refine candidate hits.

The terms are directly stored using a keyword dictionary as described in Section 2.4.1 because it is beneficial for most workloads as already shown in Section 3.1.2 for text-only indices. Hence every term is directly converted to a fixed length identifier, simplifying and speeding up many tasks.

For all returned cells, the keyword check is immediately performed, independent of their type. Afterwards, for all border cells with matching keywords, a point-in-polygon test is performed. Since the point-in-polygon test is more compute-intensive than the keyword containment check, the baseline implementation may be faster than standard geo-spatial ACT.

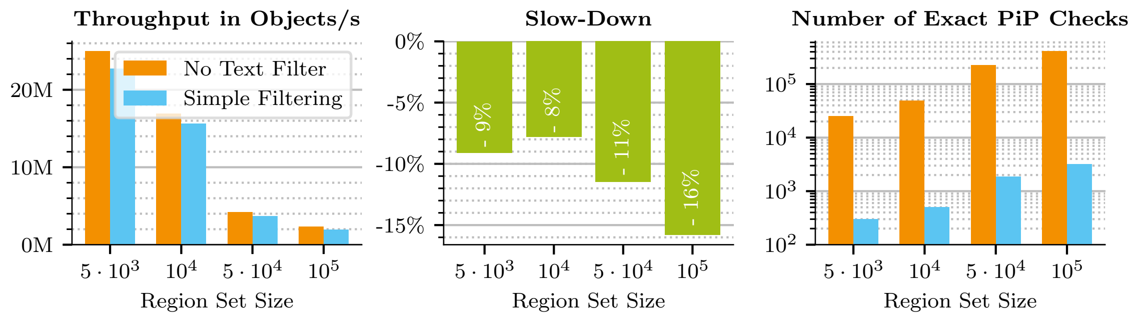


Figure 3.8: Performance Comparison with and without Simple Term Filtering

Figure 3.8 depicts the runtime difference between an unaltered ACT implementation and an ACT with additional term check. Though the index is not yet optimized, the performance difference is not severe. Both tests do not materialize the result set and do only count the matches.

When looking at the number of performed point-in-polygon tests, there is a clear difference. Since the textual selection is performed before the geospatial check. This leads to a speed-up of up to 16%.

3.2.2 Pre-Selection by Bloom Filters

As already shown in Section 3.1.3 for AKI, Bloom filters have the potential to massively reduce calls at the cost of some additional instructions and minor memory overhead. ACT has three different types of nodes with attached payload, where filters may be integrated.

Plain lookup tables are used for cells with more than two attached regions. Hence all items have to be traversed and their terms checked for containment. This resembles the situation in AKI for non-frequent cells. Thus along with each 32-bit key to the region, a 32-bit filter is stored. It is first tested for a potential match in the filter and afterwards the actual match is performed.

This configuration is first tested in a reduced text-only environment, comparing three different multi-keyword index implementations. The first implementation stores all terms as they are, without sorting them, requiring a normal subset containment operation. The second implementation uses sorted term lists allowing optimizations. The third data structure uses additional Bloom filters for pre-selection.

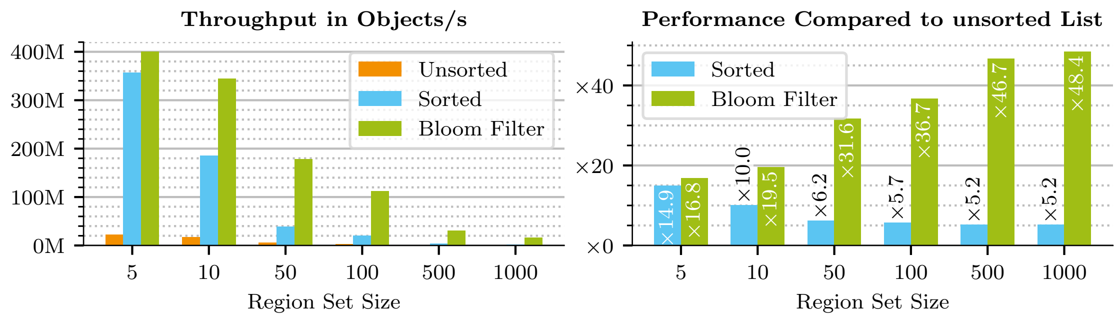


Figure 3.9: Performance Comparison of different Lookup Tables

As depicted in Figure 3.9, sorting provides a huge performance boost. Because the terms are sorted, subset containment can be evaluated in $\mathcal{O}(n)$ with n being the length of the shorter list. For unsorted terms $\mathcal{O}(n \cdot m)$ steps have to be performed for a cross product of the n respective m terms for regions and objects. Actually sorting the terms is asymptotically faster than the cross product. Sorting can be done in $\mathcal{O}(n \log(n))$.

A fast pre-filtering function additionally speeds up the computation because it reduces the number of redundant function calls. The benefit of filtering a plain list is even bigger than filtering in AKI since AKI also includes pruning steps while list traversal does not.

Figure 3.10 depicts the runtime comparison with the baseline implementation. For a small list size, ACT matches marginally slower, but with more elements the benefits clearly dominate. The more queries are indexed the higher is the probability of more than two regions sharing a cell.

This optimization only provides improvements for lookup tables, but nodes with inlined payloads are also common. Both inlined payload nodes are constructed similarly. Thus a single payload node has one unused slot for 31 bits of payload. This slot is used to store 31 bits of the associated region's Bloom filter. Hence this information does not need additional space in memory.

The rightmost bar in Figure 3.10 shows the performance of ACT with Bloom filters in the lookup table and the single payload. It clearly outperforms the baseline and is also faster than

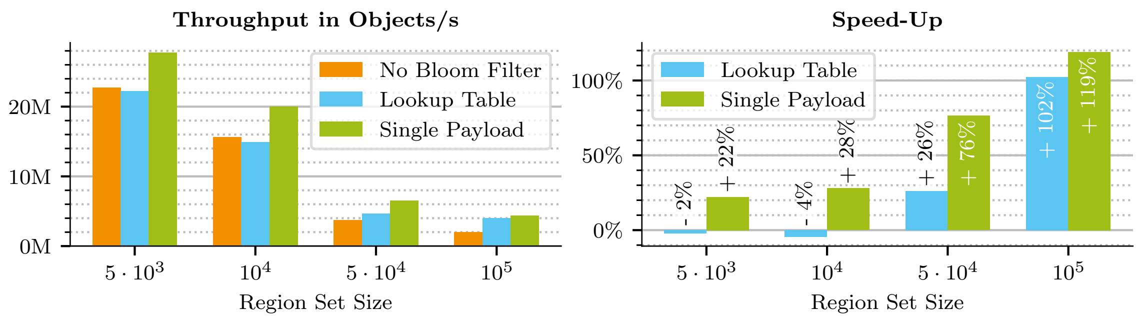


Figure 3.10: Performance Comparison with Bloom Filtering

ACT without textual discrimination.

Term containment checks reduce the amount of required exact tests for border cells massively. The number of containment checks is decreased as well by Bloom filters.

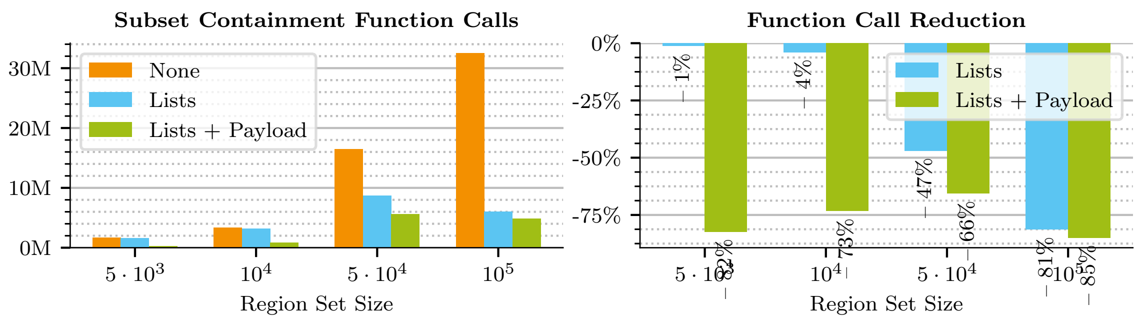


Figure 3.11: Subset Containment Call Reduction with Bloom Filtering

3.2.3 Replacing Large Lookup Tables by Textual Index Structures

The previous section already massively sped up the processing of lists in comparison to plain lookup tables. However, cases with a large number of overlapping cells have to be handled differently because they are typical for spatio-textual joins. This section handles the overlap with the optimized multi-keyword index implemented in Section 3.1.

Figure 3.12 evaluates the performance of the optimized AKI against the Bloom filtered sorted list. Both uniform and naturally skewed datasets are used. The trend for both datasets is similar. For small regions, the sorted list is faster since AKI provides a comparatively huge overhead. However, sorted lists slow down much faster than an AKI with an increasing amount of indexed regions.

Thus there is a point when switching between both structures makes sense. Depending on the workload, this point is at about 100 to 500 indexed regions. To be conservative 100 is chosen for the implementation. As a result, the payload now has five different types which cannot be distinguished using two bits.

Hence the offset payload needs the third bit to differ between an offset and a pointer to AKI nodes. Since all pointers are aligned to 64 bit, the third bit can safely be used for differentiation.

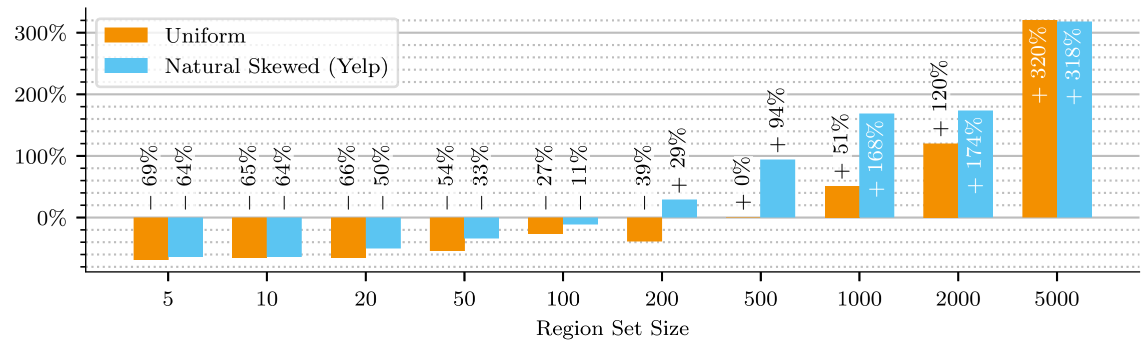


Figure 3.12: Performance of Bloom Filtered AKI compared with Bloom Filtered Sorted List

This extra bit to test adds some little overhead when testing for shorter lists.

As described in the following section, ACT has problems indexing workloads with many overlapping regions. Unfortunately, these workloads are the only ones requiring this optimization. Thus the benefits cannot be measured in an optimal way. However, Figure 3.13 depicts measurements with the clustered Yelp Dataset.

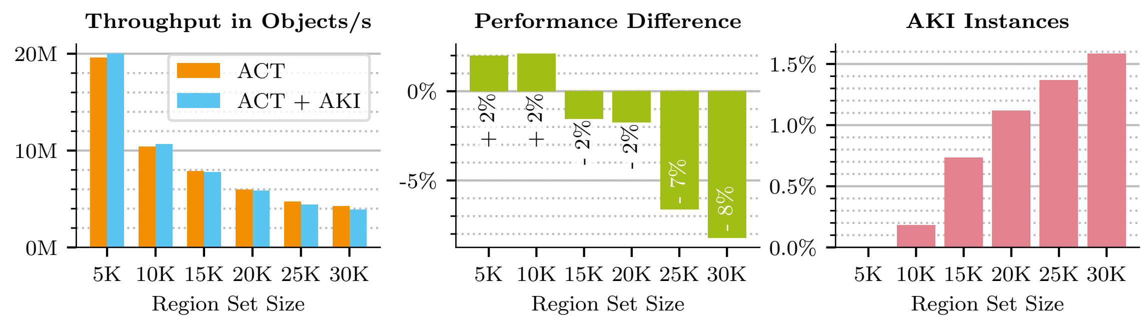


Figure 3.13: Performance of ACT with and without AKI for cells with more than 100 Regions attached

The rightmost subfigure shows the number of AKI instances which is constantly increasing with the number of indexed regions as expected. The more indexed cells are overlapping, the more AKI instances are generated. Unfortunately, the increasing amount of AKI instances does not result in the expected speed-up. The AKI variant almost slows down constantly with increasing number of AKI instances.

Though the concrete reason for the slow down is not known, a possibility could be the different way of storing the data. While the term lists are all combined into a single list only storing the offsets, each AKI is independent. Because it indexes a high number of terms, several cache-misses occur when traversing. This could be the reason why the whole matching process is slowed down.

3.2.4 Problems with Overlapping Regions

ACT was designed with geographical entities like states or districts in mind. Such administrative boundaries usually only adjoin each other and do not overlap. Hence the build process of ACT is not optimized for workloads with multiple overlapping polygons.

In contrast, subscription indexes as already shown for example in Figure 1.5 feature clustered data with extensive geographic overlapping.

The geo-spatial structure of ACT basically features a radix-trie which indexes the cells as leaves. Hence larger cells are split and replicated subsequently to the lowest levels as illustrated in Figure 3.15. The higher the overlap the higher is the number of replicated cells.

In its standard configuration, ACT divides each polygon in approximately 256 border and 256 interior cells using 128 covering cells as an intermediate step. As shown in Table 3.1 the amount of generated S2Cells is independent of the clustering because each cell is processed separately.

Polygons	**10,000**		**50,000**		**100,000**	
Dataset	**Uniform**	**Clustered**	**Uniform**	**Clustered**	**Uniform**	**Clustered**
Covering	912,839	862,758	4,564,642	4,302,297	9,136,355	8,608,920
Border	2,421,761	2,207,841	12,089,786	11,024,062	24,194,396	22,048,664
Interior	2,565,481	2,558,176	12,827,393	12,793,118	25,654,986	25,588,121
Covering/Poly	91.28	86.28	91.29	86.05	91.36	86.09
Border/Poly	242.18	220.78	241.80	220.48	241.94	220.49
Interior/Poly	256.55	255.82	256.55	255.86	256.55	255.88
Cells/Poly	498.72	476.60	498.34	476.34	498.49	476.37

Table 3.1: Comparison of Cell Count for Uniform and Clustered Yelp Datasets

Figure 3.14 shows the size of one of the internal data structures used during the build process of ACT. One statistic shows the number of distinct payloads which are cached to attach to cells. The more the data is clustered and thus overlapping, the faster the variety of payloads increases. The measurements were performed until 128GB of main memory were not sufficient. This point is marked by a red cross in Figure 3.14. At the time of the crash, the number of distinct payloads greatly varies, so it cannot be the main reason for overfull memory.

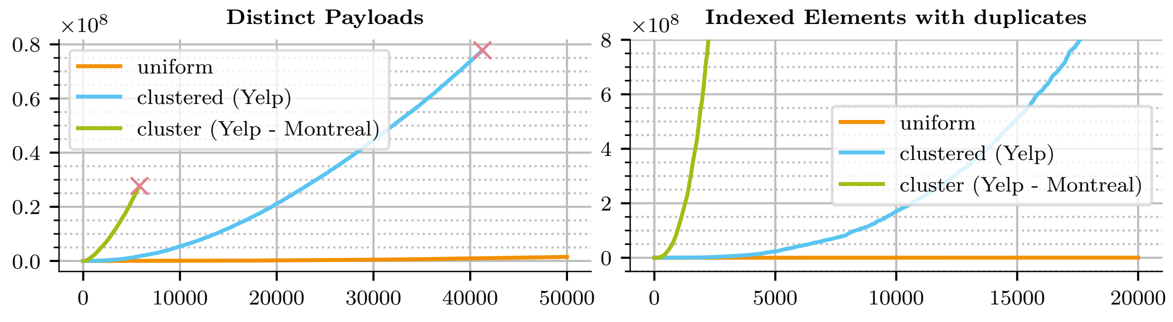


Figure 3.14: Statistics of intermediate building data structures

The right-hand side of the figure shows the summation of all elements in the payload lists.

All elements are unique inside each list but several lists can contain the same element. In this plot the number of elements rises much faster for the densely clustered Montreal dataset, leading to massive memory consumption for the lists. In both plots, the barely overlapping uniform data is several orders of magnitude lower because nearly no cell replication is triggered by overlap.

A possible solution to reduce the massive memory amount is adding an extra field for each trie node containing the cells with the identical extent to the node. This stops the propagation by effectively limiting the replication to a certain factor because it allows information stored at inner nodes. This idea is illustrated on the right-hand side of Figure 3.15.

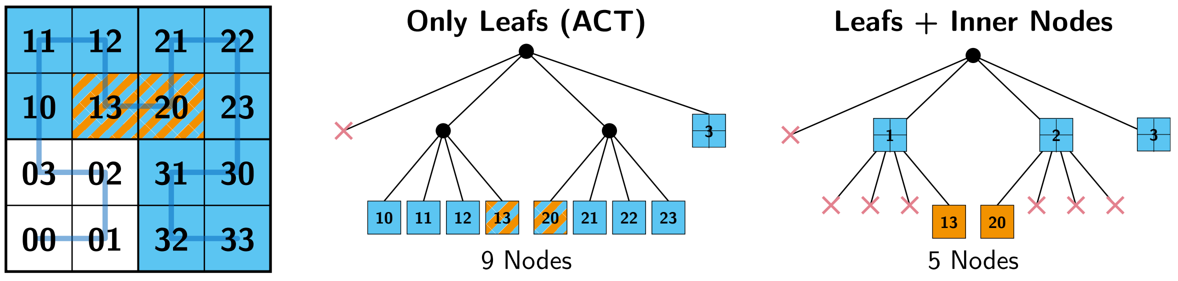


Figure 3.15: Comparison of Cell Replication for AKI using Trie or Hash Table

Since this involves modifying the build process of ACT and many internal data representations, it was not in the scope of this work.

In conclusion, ACT provides superior fast performance for up to medium-size polygon data, but it still cannot handle overlapping data. When trying, the intermediate results consume too much memory, hence the trie cannot be built. Thus ACT as its described by Kipf et al. [Kip+18] is not suitable for a typical subscription index workload.

4 Data Generation

The implemented spatio-textual data structure has to be evaluated in order to be compared with competitors like the FAST index. Since this index solely supports rectangular regions, the test dataset has to be enhanced.

The methods for dataset generation are presented in the following sections. Three different datasets are generated which feature different distributions.

Unless otherwise stated, all work in this chapter was done using Jupyter along with the SciPy ecosystem [J+01] for advanced mathematics functions and visualization.

4.1 Extruding Points to Regions

Most datasets just consist of point data, which is easier to generate than rectangles or polygons. Hence to perform spatio-textual joins, a method for transforming points into regions is necessary.

Two different kinds of approaches are presented which mainly differ in the resulting overlap and coverage of the total domain. The first approach extends each point for its own, generating possibly overlapping features. The second one partitions the space based on the input points and generates a covering plane of polygons.

4.1.1 Individual Extruding

When generating rectangles as shown in the left of Figure 4.1, latitude and longitude extent have to be generated. Minimum and maximum extent is given in meters and simply converted into extent since degrees of longitude are not consistent in length.

Optionally only one extent is given and all rectangles are approximately quadratically shaped. This requires to take the current latitude into account.

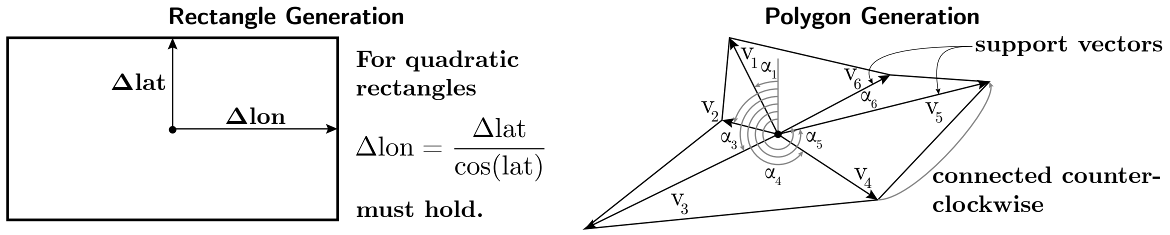


Figure 4.1: Generation of Rectangles and Polygons around Support Point

Generating random polygons is a bit more involved and can be controlled by setting the minimum and maximum for diameter and number of edges. For each edge, a length and an angle are chosen randomly. All resulting vectors are sorted based on their angle and their tips are connected to a polygon as shown in the right part of Figure 4.1. This generates random polygons which have to contain the midpoint and do not intersect itself. The size can be adjusted by varying the number of supporting vectors and their length but in contradiction to the rectangles, no minimum size can be guaranteed.

4.1.2 Domain Covering Polygons

Geospatial indexes like ACT are mainly used to join points against administrative areas. Since these polygons partition the space, they do not or only slightly overlap.

Hence simulating this workload with a high amount of regions is desirable. Zacharatou et al. present a method to efficiently generate synthetic datasets with the desired characteristic [Zac+17]. Random points are generated in a specific region. Based on these points, Voronoi cells are generated.

Random edges of the cells are deleted, joining adjacent cells. Figure 4.2 shows a cutout of the resulting polygons. They are randomly formed, differ in size and may be non-convex. Thus the entire domain is filled with an irregular shaped pattern.

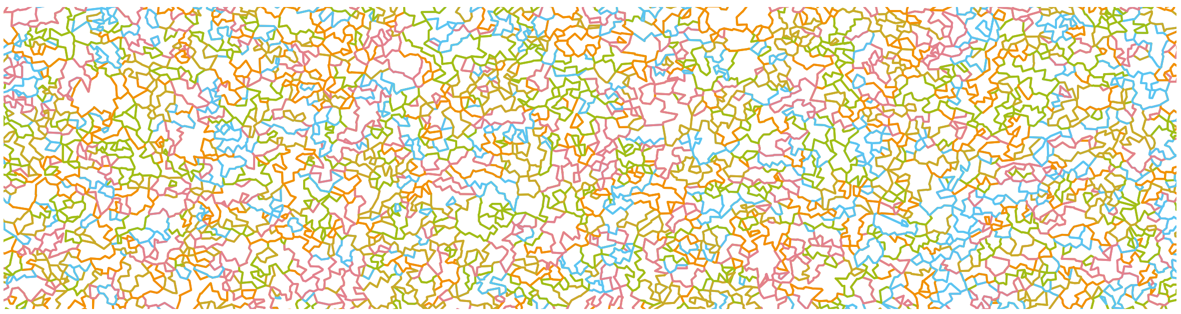


Figure 4.2: Irregular domain-filling Polygons using joined Voronoi Cells

The implementation[1] can generate a variable number of polygons in an arbitrary rectangular region. Weighing those factors, the average size of the shapes can be controlled.

Furthermore, a custom set of points can be processed to generate skewed datasets since varying point density affects the polygon size in the region.

4.2 Uniform Data

Regular data over the whole domain provides a simple yet challenging data characteristic for an index structure. It likely does not resemble the characteristics of a real-world dataset. Hence some special optimizations may be not applicable for this type of data.

Beyond that, uniform data can be generated comparably easy and fast. It does neither require prior knowledge nor particular distributions. Thus a uniform dataset consisting of textual and different geospatial data features is generated.

The term generation is parameterized to allow a fast change in dataset properties. The vocabulary size, the minimum and maximum length of the term list and the term itself are adjustable, splitting the process into two parts.

[1]Code for Voronoi based polygon generation was provided by [Zac+17].

First, a vocabulary list is generated. Its word length varies to achieve typical characteristics for variable length string comparisons. The length is randomly chosen with respect to the parameters. Each word does only consist of a combination of lower case letters.

Afterwards, the requested number of terms is randomly chosen from the vocabulary and stored as a sorted list. All required used random numbers were generated using uniform distributions. It is advisable to choose a higher length of the term list for the objects than for the subscriptions since the latter must be the superset.

Points must be generated for all spatial features. Based on an adjustable region, points are uniformly generated. When generating objects, the points are directly used and joined with the generated term lists. For regions, the generated points are extruded with the methods presented in Section 4.1.

4.3 Naturally Distributed

Uniform data does not typically resemble a typical real-world workload. To achieve a more realistic test dataset, collected data is analyzed and synthetic data is generated using the data or observed frequencies and distributions.

Since most textual data is not connected to spatial features and vice versa, two datasets are artificially combined together. This method is also used by other publications, for example by Chen et al. [Che+13], to create a synthetic spatio-textual dataset. This thesis uses location data from Foursquare in combination with a subtitle dataset.

4.3.1 OpenSubtitle Database

A comprehensive free word list dataset is provided by OpenSubtitle [LT16]. It contains about 337.8 million english sentences from unofficial translations of movies, tv shows, and similar productions. The data was post-processed by H. Dave into a word frequency list and is publicly downloadable [Dav16].

Analyzing the frequencies of the terms, as shown in Figure 4.3, they are following a Zipfian distribution. This is characteristic for real-world text data. [Pow98] It can be seen that a small amount of terms is frequent while the majority of the terms does not occur often.

Illustrated in the right part of the figure, the most frequent 100 terms occur as often as the less frequent $100,000$ terms.

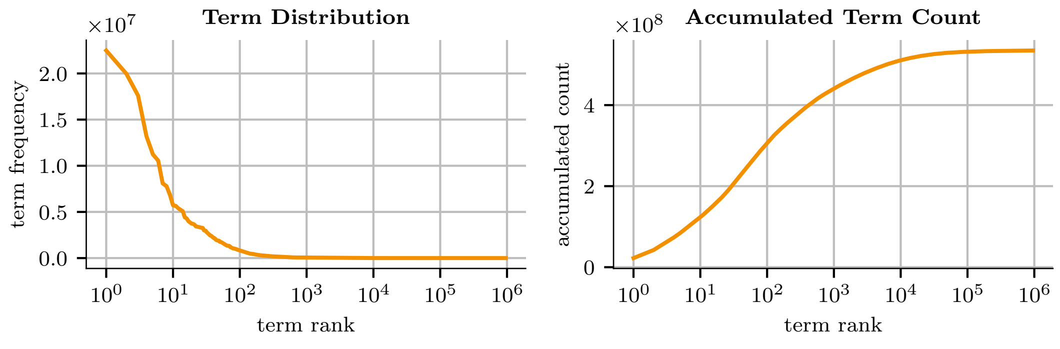


Figure 4.3: Zipfian Distributed TV Subtitles Data

The term distributions are inserted into a discrete distribution and a random amount of terms is drawn for regions and objects. In contradiction to most of the dataset generation code, this part was done in C++ since the performance for discrete distributions is superior to Python.

4.3.2 Foursquare Inspired

Foursquare check-ins correspond to places where people regularly use their mobile phones. Hence they correspond to an expected usage scenario for subscription indexes.

The used dataset was collected by Yang et al. from April 2012 to September 2013. It contains 33,278,683 check-ins on 3,680,126 points of interest. Those are clustered in the 415 cities around the world, shown in Figure 4.4. They gathered data from most of the national capital cities, all marked in orange, and a high number of provincial capitals which are marked in blue.

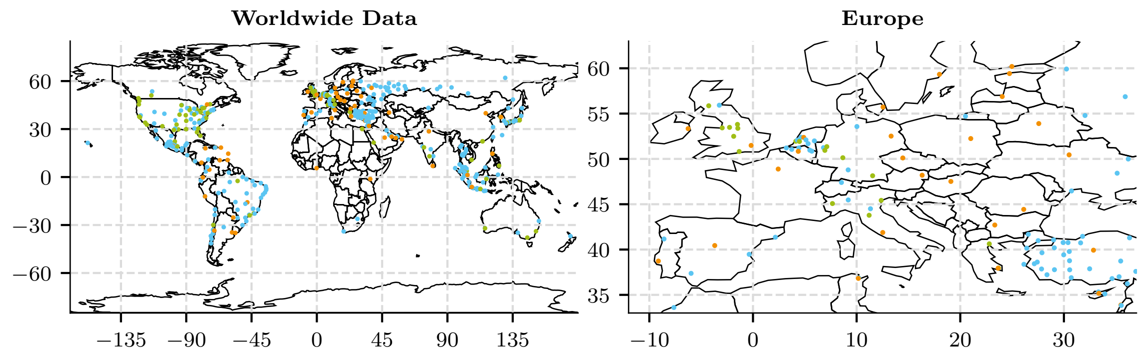


Figure 4.4: Cities in Foursquare Dataset

The data was collected using the public Twitter API filtering for automated Foursquare check-in tweets. Based on the associated location id, points of interest and city data was fetched from the official Foursquare API [YZQ16].

The points of interest, also called venues, represent the meeting spots where users can check in. Hence they are used as regions and extended into rectangles and polygons as described in Section 4.1. Because users directly checked in into this place, a comparably small range of 50 to 100 meters is chosen.

Because the check-ins itself do not contain geographical positions, the data has to be joined against the venues. This results in all check-ins of the venue having the same assigned position. To scatter the points in the locality around, it is shifted by up to 50 meters in a random direction, resulting in a point cloud for each venue.

4.4 Yelp Dataset

Yelp Inc., an online service to find and rate businesses, offers a dataset free to use for personal, educational, and academic purposes. It contains a subset of their businesses, reviews, and user data to use for machine learning, learn about databases or adapting it for other use cases [Yel18b].

For this work, the information about 174,567 businesses in 11 metropolitan areas is used. Each business is stored as JSON object holding the name, coordinates in latitude and longitude, and some categories.

The restaurant data is used to create regions with the categories as keywords. The objects for the join have to be generated with respect to the regions to match the data and generate a reasonable amount of hits.

4.4.1 Expanding Business Locations to Regions

Each business has an address and the corresponding geographic position assigned to it, making it a point feature. Because it needs outreach, the point has to be extended into a region as described in Section 4.1.

When generating rectangles, the star rating is taken into account to influence the size of the region. The higher the approval of the business is, the higher it's reach, the larger the area. All areas are generated in a quadratic shape.

Polygons are generated using a varied amount of supporting vectors starting at the business points. The number of vectors is randomly varied while the length is, as for the rectangles, controlled by the rating.

4.4.2 Generating Visitor Locations

Since the data is clustered in metropolitan areas, this fact has to be considered for data generation. Unfortunately, the corresponding area is not stored within the dataset, so it has to be determined. Hence all points are clustered using K-Means[2] with 11 clusters, euclidian distance, and K-Means++ for initialization of cluster centers.

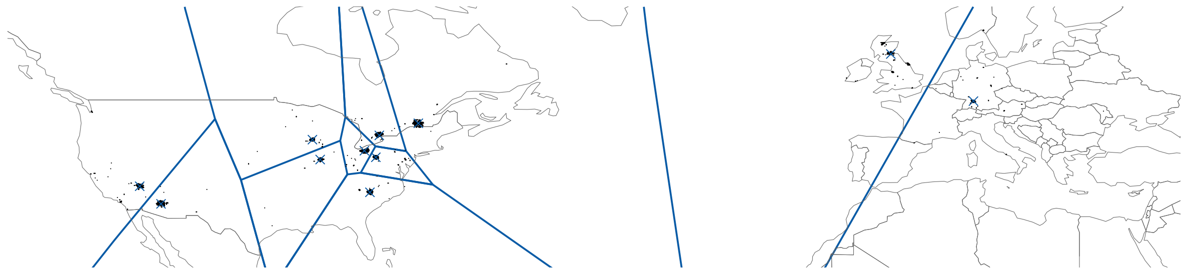


Figure 4.5: K-Means Clustering on Yelp Data

The resulting clustering is visualized in Figure 4.5. The data domain was limited to the shown area of USA and Europe because outliers heavily affect the results of K-Means. In total 174,522 businesses were classified into areas. The count of businesses, along with the calculated cluster centers, is shown in Table 4.1.

After obtaining the 11 clusters, parameters have to be extracted to generate points in the respective area. The point sets are analyzed determining mean and covariance. Due to distant outliers, the computed covariance matrix indicates a comparably high variance. This is illustrated as error ellipses for one to three standard deviations in the left subfigure of Figure 4.6.

[2]Implementation was provided by Scikit-learn. [Ped+11]. The enlisted parameters are the default configuration.

Metropolitan Area	Business Count	Center Latitude	Center Longitude
Phoenix	52204	33.492	-111.989
Las Vegas	33104	36.122	-115.176
Toronto	30201	43.707	-79.429
Charlotte	13632	35.215	-80.832
Cleveland	12610	41.418	-81.655
Pittsburgh	10106	40.441	-79.950
Montreal	8196	45.513	-73.611
Madison	4763	43.074	-89.409
Edinburgh	4687	55.852	-3.135
Stuttgart	3165	48.821	9.194
Champaign	1854	40.105	-88.252

Table 4.1: Number of Businesses by Metropolitan Area

In order to reduce the variance of the point clouds, a simple outlier filter is applied. The most distant 3% of the point with respect to their cluster center were not considered for calculating covariance leading to better results shown in the right part of Figure 4.6.

This can be seen best when looking at Champaign and Pittsburgh. While the variance is among the greatest in the original dataset, the error ellipse showing Champaign's variance is barely visible on the right-hand side.

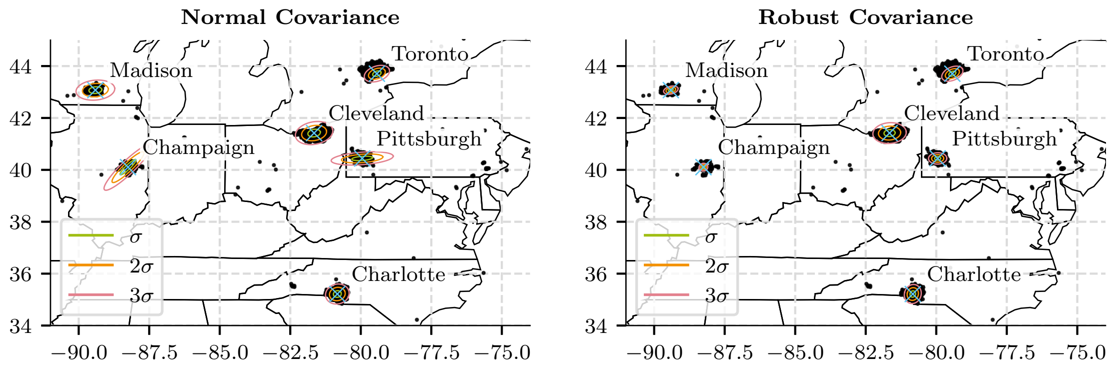


Figure 4.6: Difference between Normal and Robust Covariance

Mean and covariance matrix specify a 2-dimensional multivariate normal distribution and can be used to generate points in the desired area. For this purpose, a program capable of generating an arbitrary number of visitor locations was developed using Eigen.[3]

Since the count of businesses in each area vastly varies as shown in Table 4.1, this is also taken into account. A discrete distribution approximates the point's distribution to the restaurant's distribution.

[3]Eigen is available at `http://eigen.tuxfamily.org`

4.4.3 Generating Term Lists

For the moment, only spatial features can be produced, but the fitting textual features have to be generated as well as to be combined into a valid object. The terms have to be analyzed to generate better sets than random. Weighted random does only provide reasonable results for independent distributed datasets.

Using the spatial clustering, the term lists are also grouped by metropolitan area. As seen in Table 4.2, regional categories like *German* or *Swabian* are common. Thus each of the cities term lists is processed individually.

At first, each object's terms are indexed into a graph,[4] storing the relationships between associated terms. An edge is added to the graph for each list element's permutation with a size of two. In case the edge already exists, the weight of the edge is increased. Hence the edge weight stands for the number of connections from node to node. For example, if *Restaurant* to *Italian* has a weight of 50, this number of term lists contain both words.

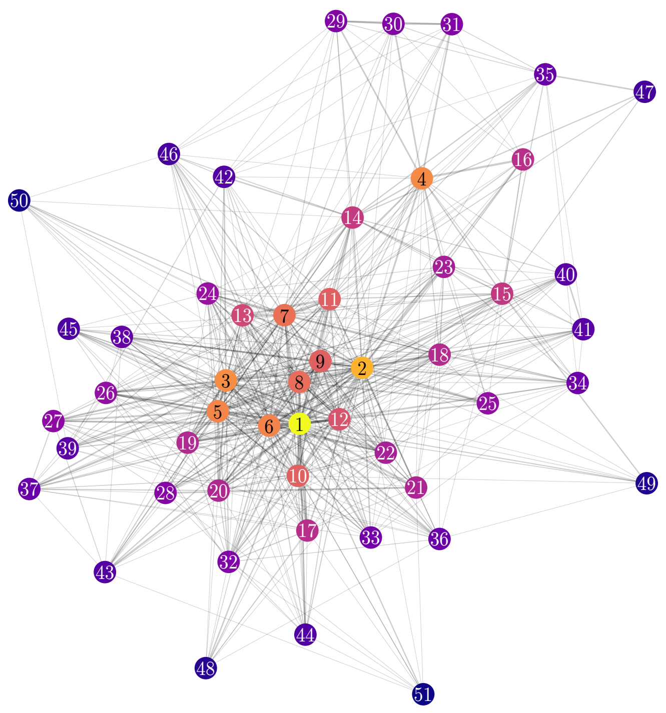


Figure 4.7: *k*-Core with degree 21 of Keyword Graph from Stuttgart

#	Category
1	Restaurants
2	Food
3	Nightlife
4	Shopping
5	Bars
6	German
7	Event Planning & Services
8	Cafes
9	Coffee & Tea
10	Italian
11	Hotels & Travel
12	Swabian
13	Hotels
14	Arts & Entertainment
15	Active Life
16	Fashion
17	Pizza
18	Specialty Food
19	Wine Bars
20	International

Table 4.2: Top 20 Categories in Stuttgart

The core of the graph for the area of Stuttgart is depicted in Figure 4.7. Stuttgart was chosen since it featured a comparably small vocabulary size resulting in a more overseeable graph. Furthermore, only the k-core with $k = 21$ is shown. The k-core is the maximal subgraph where the degree of all remaining vertexes is at least k. It is obtained by repeatedly deleting all vertices which a degree less than k following by the increase of k until the graph would be empty.

The color and numbering of the vertices indicate the total accumulated weight of all

[4]Graph handling, visualization and analyzes were done using NetworkX [HSS08].

outgoing edges. Thus it stands for the total frequency of the term. Both edge weight and color are displayed logarithmically since the terms follow a Zipfian distribution. [Pow98]

A C++ program converts the graph's frequency data into probabilities which are used to generate term lists. For this purpose, a start node is drawn weighted with regard to the total frequency. Beginning with this node, based on the edge weights, the graph is explored to find a likely contained subgraph.

The exploration is either stopped when the number of terms reaches the required threshold or when the subgraph is maximal and cannot be expanded further. In the case of the latter, a new search is started at the start node and the resulting terms are joined in the end. This is repeated until the threshold is reached or the whole neighborhood is exhausted.

5 Evaluation

The developed data structures will be evaluated against their respective competitors. While the spatio-textual ACT is only tested against its main competitor, the FAST index [MAA17], several competitors were implemented to challenge the optimized AKI. It is compared with most data structures presented in Section 1.4.2.

Preceding the tests an overview of the used datasets is given, of which most of their creation was described in Chapter 4. Hence only a short overview of their core parameters is given with some visualizations.

5.1 Datasets Overview

Several datasets were used for testing the data structures. Each dataset consists of two parts: One part holds the regions while the other holds a stream of objects for the test. To show the characteristics of each dataset, two plots were made per set to compare their spatial and textual features.

The spatial plots feature a binned representation of the regions' center points, respective the object's location. A logarithmic colormap is chosen to visualize the different bucket counts. The displayed region aims to show all points, only some outliers might be cropped.

For the textual plots, two distributions are shown side by side. On the left side, the relative frequency is shown for both region's and object's terms. The terms are sorted according to their frequency in the regions set. Hence differences in rate can be spotted. On the right-hand side, the accumulated distribution is plotted, in order to give an overview of the whole distribution.

5.1.1 Uniform

The uniform dataset is artificially generated as described in Section 4.2. Its' spatial distribution is not shown in a plot. Points and rectangles are generated throughout the whole domain, only omitting the polar regions above $85°$ and below $-85°$ latitude. Due to the circular shape of the earth, the distance between two degrees of latitude is not constant. Thus the point density increases from the equator to pole.

This has different effects on the compared data structure. The FAST index does not recognize this change in density because the world is modeled as XY-plane without any spherical features. Only the larger size of regions in terms of latitude and longitude is noticed. In contrast, ACT using S2 which is approximating the sphere as cube will recognize the increase in density.

The textual distribution is also uniform and very similar for both regions and objects. The only recognizable difference is in list length because the objects contain more terms, to increase the probability for matches.

5.1.2 Naturally Distributed

As explained in Section 4.3, the two real-world datasets from Foursquare and OpenSubtitles were joined to form the naturally inspired test dataset. Hence each figure shows the characteristics of one of the source datasets.

As illustrated in Figure 5.1, the data is distributed across 77 countries on all habitable continents. The distribution of both regions, corresponding to venues, and objects, derived from check-ins, follows the clustering seen in Figure 4.4 showing the cities.

In general, regions and objects are equally distributed, but in some cases, for example in Turkey, minor differences occur in frequency.

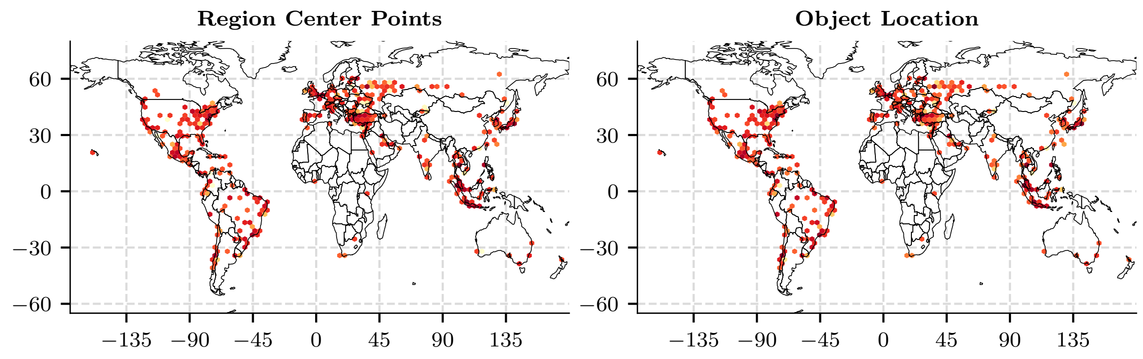


Figure 5.1: Spatial Comparison between Naturally Inspired Regions and Objects

The textual data, depicted in Figure 5.2, resembles the original distribution shown in Figure 4.3. Because both term lists are generated out of the same distribution, the graphs closely resemble each other.

The only difference occurs in a significant drop in terms of frequency for the 10 most common terms in the objects list. It is a consequence of eliminating duplicates from the lists. Since these terms are common, representing 25% of the whole vocabulary, it is plausible they are drawn two times. Because the object's lists are longer than the region's lists, they are eliminated more often in the object's list explaining the drop.

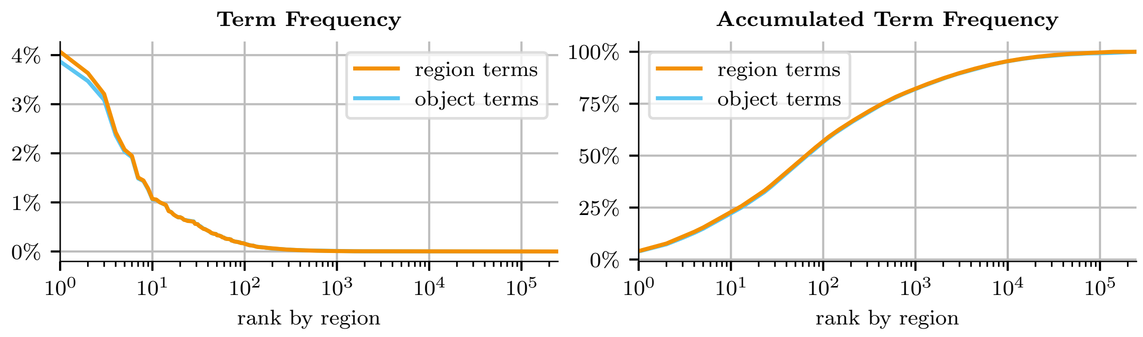


Figure 5.2: Textual Comparison between Naturally Inspired Regions and Objects

A geospatial query on this dataset is more selective by factor 2000 than a textual query.

5.1.3 Yelp Dataset

The creation of the Yelp dataset is shown in Section 4.4. It features regions directly derived from business data and objects resembling the spatial distribution and category connections.

The regions are, as depicted in Figure 5.3, mainly located in the US and Canada, while some data is also located in Europe. Close inspection reveals some outliers in the regions. Due to the artificial generation of the location of the points from clusters, outliers are not presented in the points' locations. However, the region around Edinburgh is filled with points, since the variance is comparably high for this cluster.

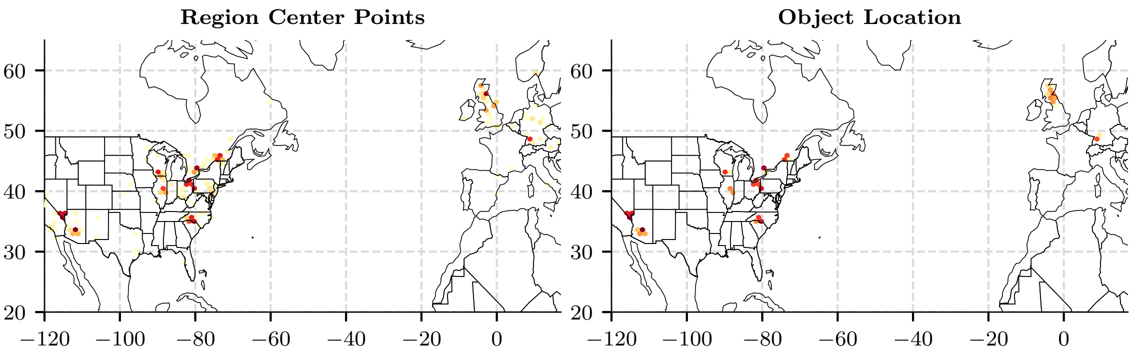


Figure 5.3: Spatial Comparison between Yelp Regions and Objects

Comparing the term frequencies, depicted in Figure 5.4, both show the same trends but locally differ. This is a consequence of the technique based on a graph used to draw the terms. Along with the frequencies, the term connectivity is taken into account. This prefers more interconnected terms and leads to the variations seen in the left plot.

The term *Shopping*, for example, takes the most frequent term over from *Restaurant* for the objects. As seen in the right graph, common terms for regions are a bit less frequent for objects.

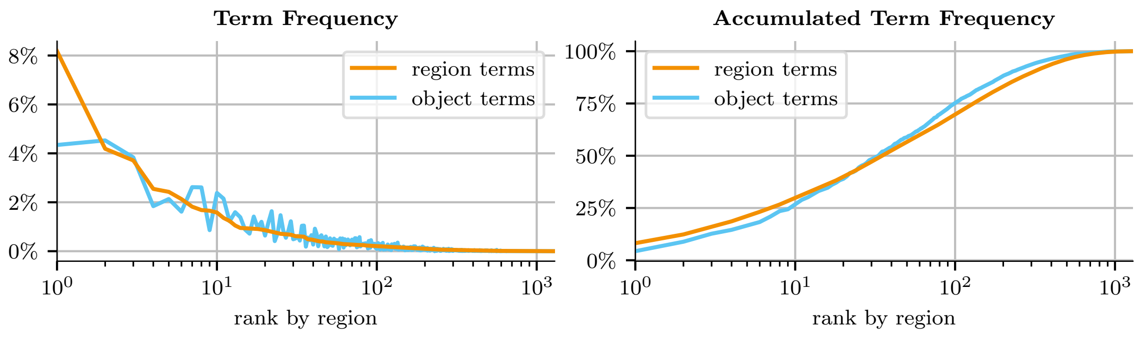


Figure 5.4: Textual Comparison between Yelp Regions and Objects

A geospatial query on this dataset is more selective by factor 4 than a textual query.

5.1.4 Twitter Dataset

The Twitter dataset was not created for this thesis. The authors of FAST used it for their original evaluation and provided it for comparison of the index structures [MAA17]. The dataset was collected from January 2014 to March 2015 from the public Twitter feed and contains about 30 million geotagged tweets around the area of the US.

Comparing objects and regions shown in Figure 5.5, one can notice a data skew to the top right corner in the regions dataset. This is induced due to the method used for converting points to regions. The lower left corner is fixed and expanded to the top right. This effect gets stronger when increasing the size of the regions.

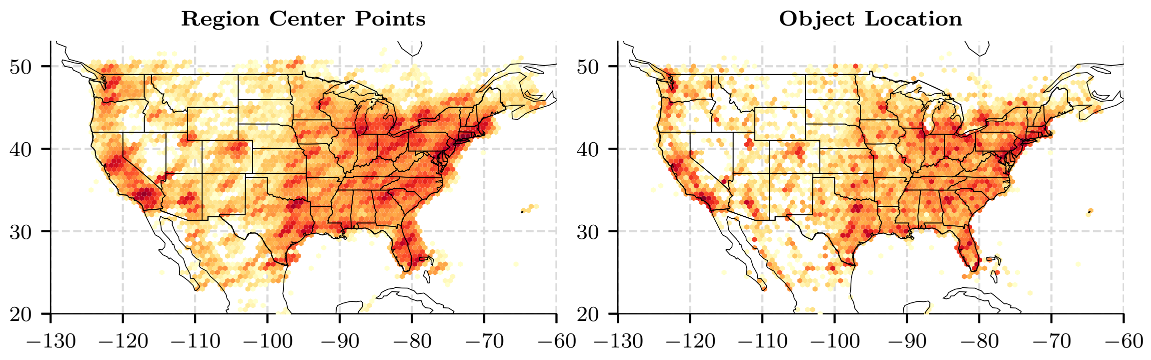


Figure 5.5: Spatial Comparison between Twitter Regions and Objects

The textual distribution of regions and objects, depicted in Figure 5.6, shows the same trends for both. But some terms are clearly more present in objects.

This follows from using a particular algorithm for data generation. A minimum length of object terms is given as input. In case the processed object does not hold enough terms, the term list of the most recent object with enough terms is appended.

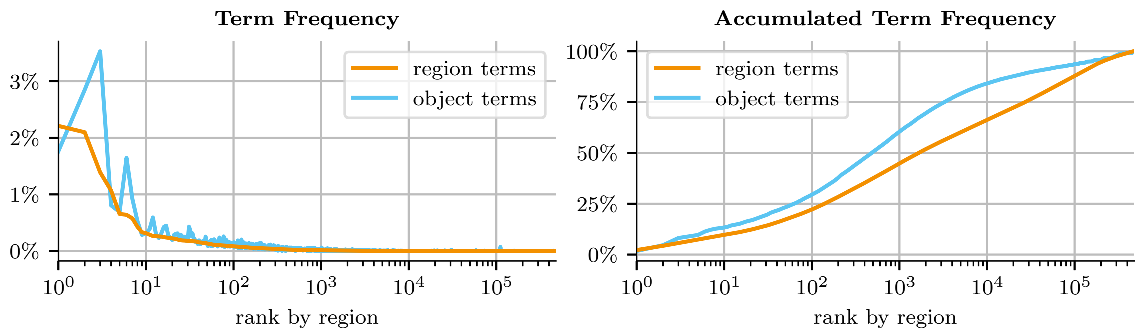


Figure 5.6: Textual Comparison between Twitter Regions and Objects

Hence terms from longer lists tend to appear more often in the data. Since all lists are stored in a sorted form, terms with initial letters in front of the alphabet are even more preferred. As depicted in Figure 5.7, terms starting with the letter *a* appear twice as often in the regions set while letter *v* is almost 50% less.

When looking at the curves in Figure 5.6, two high spikes in the ten most recent words occur. These spikes correspond to the terms *2015* and *2014* because in both years many people tweeted greeting for the new year.

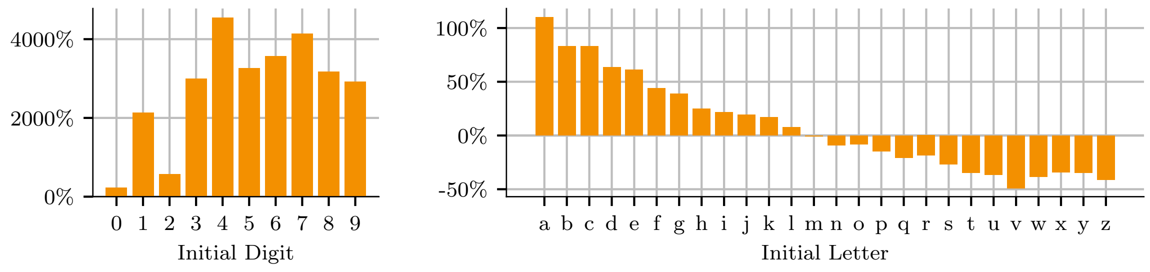


Figure 5.7: Correlation between Initial Character and Occurrence Compared between Regions and Objects

Both terms are especially common in objects because numbers are sorted in front of letters. Hence they tend to be included in many terms as depicted in Figure 5.7. Terms starting with numbers are so rarely in the regions term list despite *2* for obvious reasons. Though, they are replicated quite likely leading to for example digit *4* being 40 times more often in objects than in regions.

5.2 Experimental Setup

All experiments were conducted in a similar environment to ensure reproducible outcomes. All measurements were performed multiple times and the median value of all measurements was picked.

Workload The workload is derived from the datasets presented in Section 5.1. The object workload matches the dataset and unless stated otherwise one million objects are used.

Name	**Origin**	**Characteristic**	**Vocabulary Size**	**Count**		**Average Terms**	
				Region	Object	Region	Object
Uniform	Section 4.2	uniform	10,000	arbitrary	arbitrary	≈ 2.5	≈ 4.5
Natural	Section 4.3	415 clusters	125,229	3,680,126	33,278,683	≈ 2.5	≈ 4.5
Yelp	Section 4.4	11 clusters	1294	174,567	arbitrary	≈ 3.8	≈ 8.8
Twitter	[MAA17]	skewed	945,676	30,000,000	100,000	≈ 3.0	≈ 2.9

Table 5.1: Key Figures of the used Workloads

Parameters All necessary parameters are given to reproduce the measurements at the respective place. The split threshold for AKI is set to 5 as proposed in the paper [MAA17].

Testbench Setup All analyses for text-only indexes were performed using an Intel Core i7-7700HQ with a base tact of 2.80 GHz and 16GB DDR4 RAM running Ubuntu 16.04.5 LTS. Only the single core performance was measured using the processor's turbo frequency of up to 3.8 GHz.

For the measurements of spatio-textual indexes a machine with Intel Core i9-7900X having 10 physical cores and a base tact of 3.30 GHz and 128 GB RAM running Ubuntu 18.04.1 LTS is used. When more than one core is used, the number of cores is stated in the respective figure. Single tasks profit of a turbo frequency of up to 4 GHz.

Everything is compiled using gcc 7.3 and the associated standard library implementation.

5.3 Subscription Indexes

Section 3.1 presents an efficient implementation and optimizations of the textual AKI index. The following section uses the textual parts of the previously described datasets to evaluate the index with different hash table implementations since the optimized version solely relies on the GNU standard library for providing fundamental container structures in C++.

Afterwards, AKI is compared with several competitors like OKT and RIL which structures were described in Section 2.2.

5.3.1 Hash Table Comparison

Hash tables are a key component of all presented subscription indexes. Profiling reveals that more than 50% of the processing time is actually spent executing operations on the hash table. Hence measuring index performance is an evaluation of the underlying hash table to a significant degree.

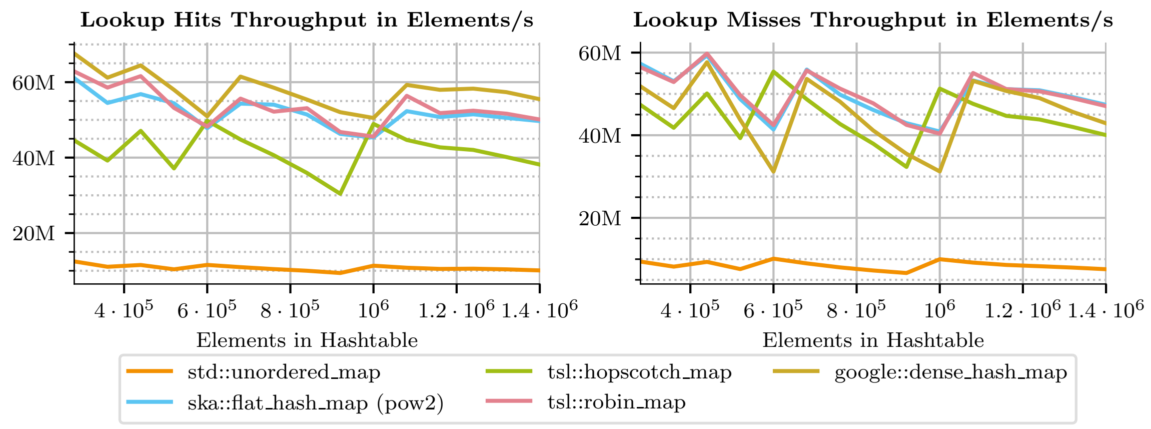


Figure 5.8: Lookup Performance of Hash Table Implementations (Data from [Tes16])

Figure 5.8 is based on a comprehensive overview of benchmarks from different publically available hash tables. It offers statistics for inserts, lookups, and deletes in various orders and for various data types. Because this implementation relies on a keyword dictionary and only the lookup is time critical, the statistics for integer lookup were considered. A choice of the best algorithms for lookup performance was plotted.

Unfortunately, Google's dense hash map [Goo16] has issues when inserting unique pointers because they cannot be moved. Thus it was not considered for the measurements. Both implementations of Tessil [Tes16] meet the requirements so an implementation of Robin Hood and Hopscotch hashing, both briefly described in Section 2.4.3, is incorporated in the test.

Skarupke's hash table is also based on Robin Hood hashing [Ska18b]. He additionally released a new version of the flat hash map depicted in Figure 5.8. It uses Fibonacci Hashing to distribute the items to the buckets. The default implementation uses powers of 2 for the capacity of the hash map to efficiently distribute the elements [Ska18a].

Thus in total four alternatives to the standard library were measured. In contrast to their implementation, all competitors use open address hashing requiring no chaining.

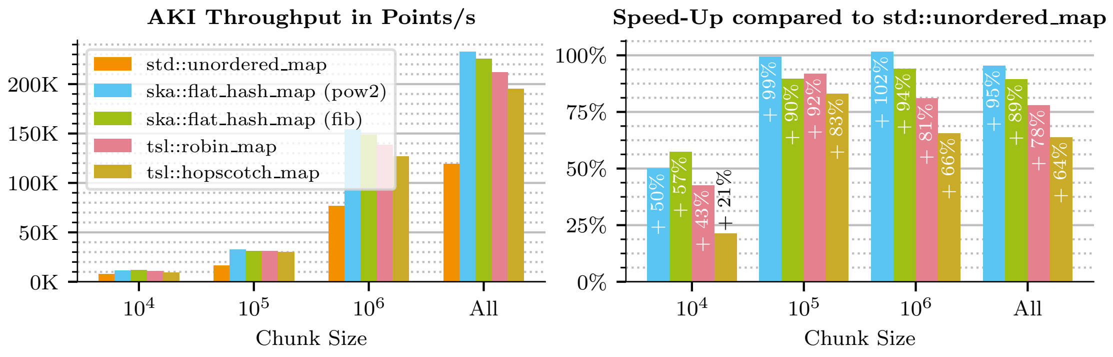


Figure 5.9: Performance Difference with different Hash Table Implementations

Figure 5.9 presents the results of the runs with different implementations used for hash tables. As conjectured, all competitors' lookup performance is notably faster than the standard library. When having small chunks the exchange does not pay off as much as for bigger workloads which have smaller overhead.

In general, the implementations of Skarupke perform slightly better than Tessil and also Robin Hood Hashing works best. As a result, the default implementation using powers of two is chosen to replace the standard container almost doubling the throughput.

5.3.2 Competitors

AKI aims to combine the strength of its both components RIL and OKT [MAA17]. These two data structures were implemented as well and tested against the optimized and unoptimized version of AKI. All four competitors feature the same interface and thus are interchangeable. Additionally, a keyword dictionary is used for all indexes, hence no character strings are indexed. Each of the four indexes used the hash table implementation of Skarupke chosen in the Section before.

Figure 5.10 shows the result of the comparison. As expected, the optimized version using bit-filters and reduced term count is faster than the original implementation by nearly a third. Ranked inverted lists perform similar nearly independent of the chunk size. Thus it is the fastest for small chunks but as the other implementations speed up, RIL stays slow.

OKT gains speed with larger chunks and is slightly faster than the optimized AKI. For small chunks, OKT has too much overhead due to the nearly empty long chains of terms so it is the slowest index. Optimized AKI performs well independent of the chunk size. It is always one of the two fastest index structures and in case it is not the fastest it is just slightly slower.

Additionally, measurements were carried out, depicted in Figure 5.11, comparing the speed up introduced by exchanging the hash table implementation.

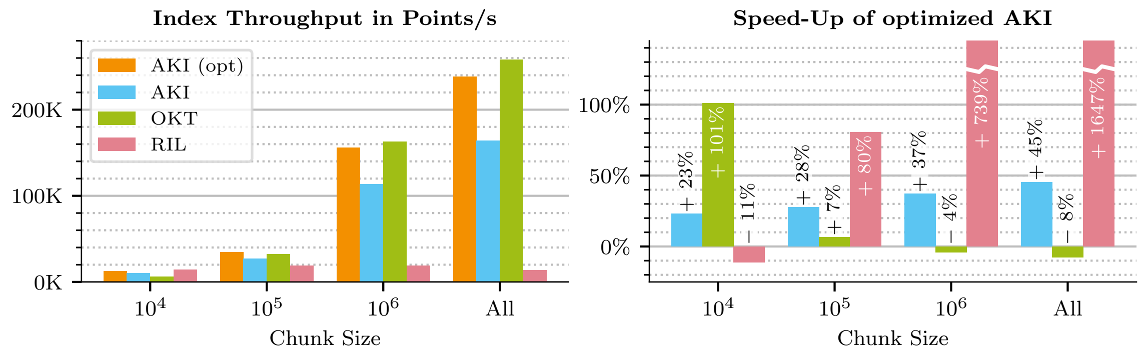


Figure 5.10: Performance Comparison of Subscription Indexes

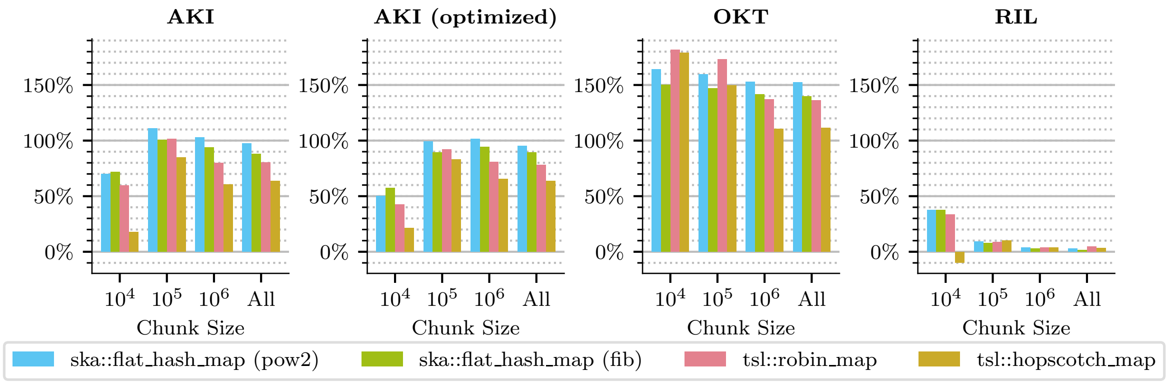


Figure 5.11: Performance Comparison of AKI Competitors with different Hash Table Implementations

Both AKI and its optimized version profit equally from the hash table change because they use hash tables equally. OKT profits significantly more since it basically is a small wrapper around multi-level hash-tables. RIL nearly does not profit.

5.4 Spatio-Textual Indexes

Previously in Section 3.2, a variation of ACT with spatio-textual features has been described. Additionally, the authors of FAST [MAA17] kindly provided their source code, testbench, and workload to be used as a competitor. Since all implementations in this thesis were done in C++, a cleaned up version of FAST was implemented using C++17.[1]

It features the same class structure and similar function signatures. The only two notable differences are that the C++ variant uses only two different payload types as explained in Section 3.1.1 and it uses the same lookup optimized hash map from the comparison in Section 5.3.1.

This section provides runtime measurements and comparisons of the enlisted spatio-textual indexes.

5.4.1 Comparison of FAST Java with FAST C++

Most measurements described by the authors of the paper were redone on the Skylake-X cluster described in Section 5.2. This features the same parameters and dataset as in the original paper. All conducted tests are summarized in Table 5.2. The default parameters are printed in bold font. The C++ code is executed single threaded to be comparable to the single threaded java implementation.

Parameters	**Values**
Frequent Term Threshold	2, **5**, 7, 10, 25, 50, 100
Spatial Range	0.01%, 0.05%, 0.1%, 0.5%, **1%**, 5%, 10%
Number of Term per Region	1, **3**, 5, 7, 9
Number of Regions	10^6, $2.5 \cdot 10^6$, $\mathbf{5 \cdot 10^6}$, $10 \cdot 10^6$, $20 \cdot 10^6$

Table 5.2: Overview over FAST Parameters used in Experimental Evaluation

The Java code was compiled using OpenJDK 11 on Skylake-X cluster. In contradiction to the FAST paper, throughput is measured to emphasize the faster data structure.

Frequent Term Threshold This parameter influences the process when a term becomes frequent and is thus handled differently by the data structure. Since this parameter is not given by the workload it can be used to tune the index. The authors of FAST propose the value 5 for the parameter since it offers a compromise between throughput and memory usage.

In this measures, the throughput of the C++ and Java version tends to decrease down for a high threshold and match the observations from the FAST paper. But five is still a reasonable choice. As Figure 5.12 shows, the C++ variant is faster but shows similar characteristics as the

[1]The FAST C++ version does not include the changes introduced in Section 3.1 like keyword dictionary or bloom filters.

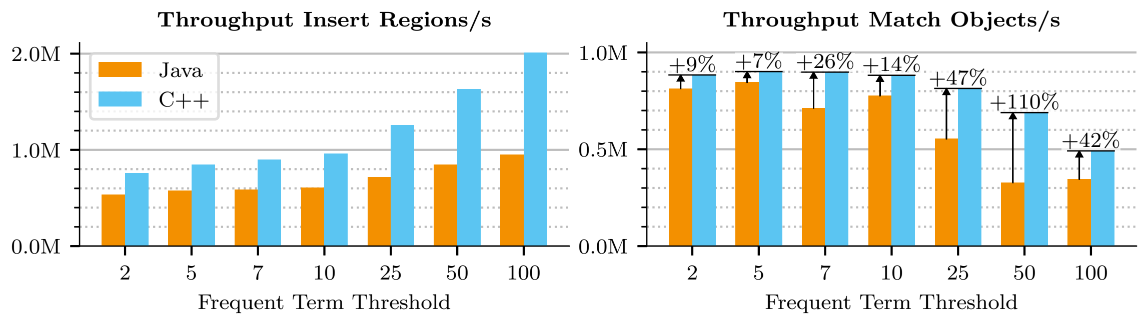


Figure 5.12: Performance Comparison under varying Frequent Term Threshold

Java implementation. However, traversing long lists in C++ is faster as high threshold do not diminish the performance as much as for the Java implementation.

Spatial Range The extent of the regions directly influence the number of generated results. Figure 5.13 shows similar results to the FAST paper. Interestingly the insert performance degrades slightly in the Java implementation with a larger spatial range which is not the case in the original measures. This could be caused by the greater extent leading to more spanning cells.

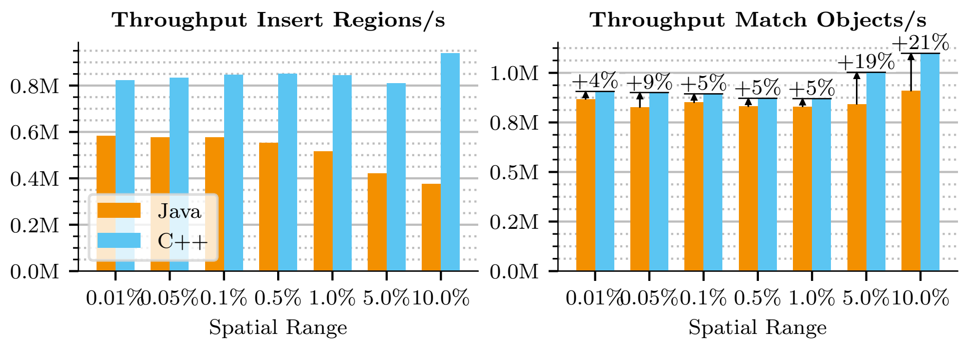


Figure 5.13: Performance Comparison under varying Spatial Range

The matching performance of the C++ implementations increases for large spatial ranges. This is a result of different error handling for faulty coordinates. While the Java version accepts too big polygons, the C++ variant rejects polygons which do not have valid coordinates. This reduces the number of indexed regions and thus results in the observable speed-up.

Number of Terms Each region has a set of terms which should be fully contained in the object terms. The region term count is altered using the method described in Section 5.1.4. The performance increases with a higher number of terms. Also, the number of results decreases because each region needs to match more queries.

Figure 5.14 shows that throughput for Java and C++ is almost equal except for one term per region, leading to many returned matches. This is the only measure, where Java is faster than the C++ implementation.

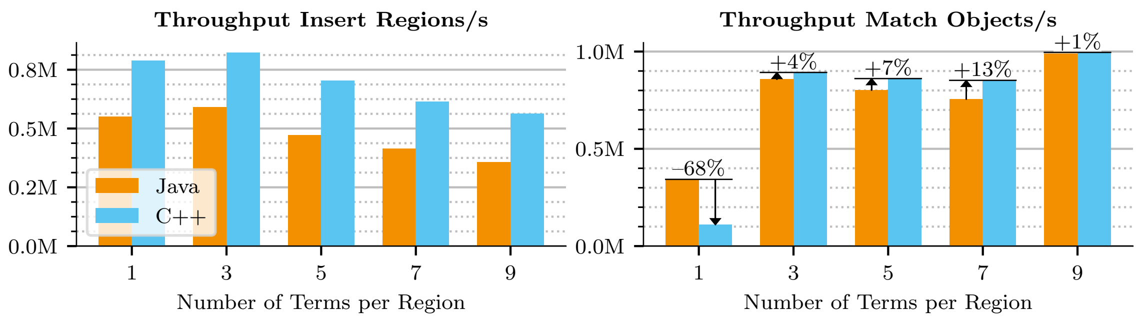


Figure 5.14: Performance Comparison under varying Number of Terms

Number of Regions FAST was tested from 1 million up to 20 million indexed regions. The matching throughput decreases slowly with the number of indexed regions because more regions need more steps to check. Eventually, the spatial pyramid also has more levels leading to more checks in the index structure.

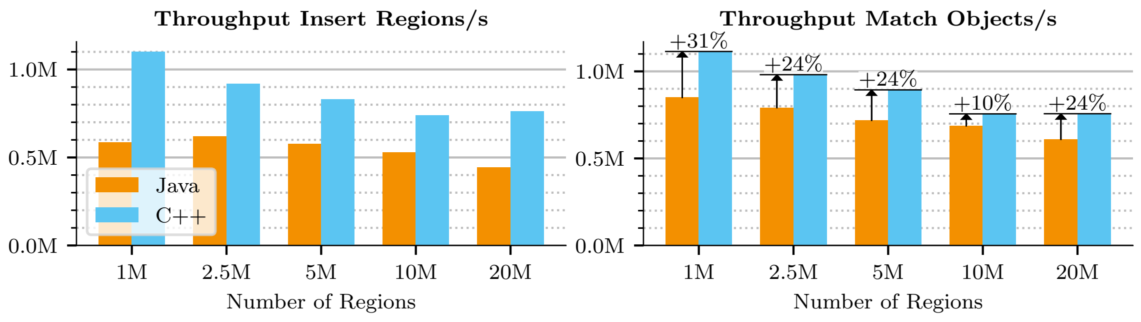


Figure 5.15: Performance Comparison under varying Number of Regions

C++ and Java perform very similarly. Their performance also degrades almost by the same amount.

Conclusion The measurements presented by the authors of FAST can be replicated using a different hardware setup. The C++ competitor provides a faster build phase and is up to 25% faster for reasonable parameter choices.

5.4.2 Multithread Performance of C++ Implementation

The C++ Implementation was parallelized using Intel TBB. The library features an easy usable API to dynamically parallelize the workload in blocked ranges. During implementation, constant functions for lookups were used. This ensures that lookups cannot change the data structure while matching objects.

Thus each lookup has no side-effects and multiple lookups can run in parallel without any synchronization. However, it has to be ensured that no inserts are performed concurrently or along lookups. Since our workload first adds all regions and queries them afterwards, parallelization is trivial.

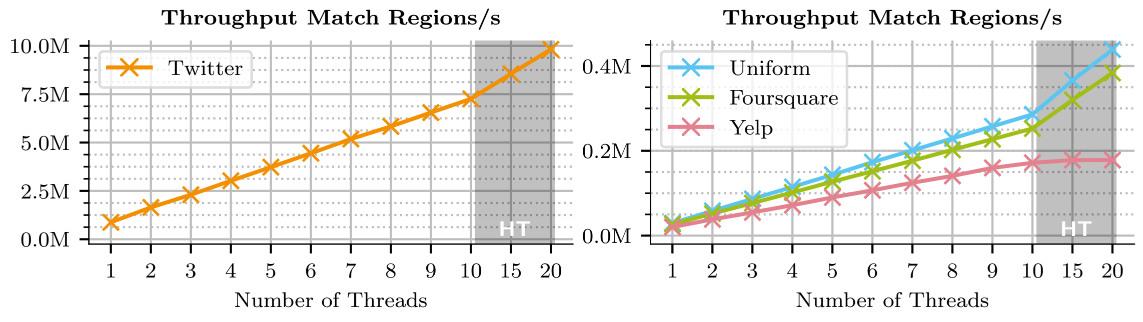


Figure 5.16: Performance when Increasing Thread Count

Figure 5.16 illustrates the speedup of the C++ implementation when gradually increasing the thread count on the Skylake-X cluster. The code speeds up nearly optimal resulting in a speedup of 7-8 when using all 10 physical cores.

When further increasing the count of threads the code still speeds up but not by the same factor. Using twice the number of threads than cores on the platform, a total speed-up of slightly more than factor 10 can be achieved. This is shown in the grey area indicating hyper-threads. Only the Yelp dataset does not profit significantly from the use of hyper-threads.

5.4.3 Parametrized Uniform Dataset

As seen in Figure 5.16, the Twitter dataset provided by the authors achieves a performance of approximately factor 10 faster than all self-generated datasets. The following section will alter the parameters of the uniform dataset to resemble this performance. First and foremost, the Twitter dataset does only produce a small number of join results. This is induced by the large vocabulary size because tweets were not meant to be joined against each other by design.

The other three data-sets produce a much higher number of results because they were generated using different techniques. The parameters of the uniform data generator are altered to reproduce some of the characteristics of the Twitter dataset.

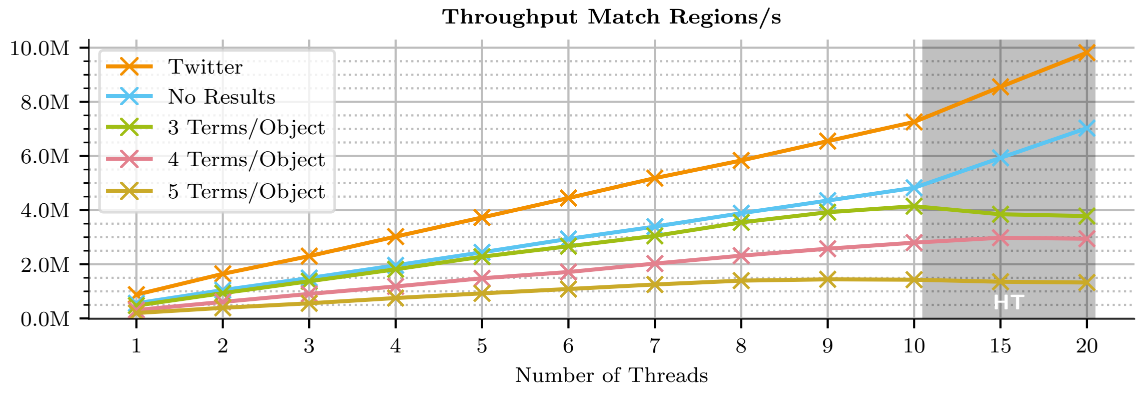


Figure 5.17: Performance when varying Selectivity

Figure 5.17 compares the performance of datasets with various characteristics. The most similar performance is achieved by a disjunct dataset leading to no results. It performs almost as well as the twitter data and hyper-threads lead to the same speed-up. It still does not reach the performance of the original Twitter data because the object's average term length is higher than in the twitter dataset.

The other contestants incrementally increase the average number of terms per object. Thus the number of comparisons and results also increases. The more terms an object has on average, the lower the performance of FAST gets.

Hyper-threads also do not provide as much benefit as they did for small result sets since the cores have to process much more when actually delivering results. Thus there is not as much spare time waiting for data retrieval which is usually used by hyper-threads.

Finally, the runtime benefits of the Twitter dataset, used by the authors of FAST for the original evaluation, can be explained as following: They are based on the low number of terms for regions and objects and the small number of found join partners.

5.4.4 Comparison of Textual ACT with FAST C++

The Adaptive Cell-Trie was designed for non-overlapping polygons. Unfortunately, the preceding workloads could not meet these requirements. Thus a polygonal workload is used to show the performance that ACT is capable of. To reduce the number of indexed cells, each polygon only produces $\frac{1}{4}$ of the usual cells to handle the high number of polygons.

Since FAST cannot index polygon data, a dataset containing the bounding boxes of the Voronoi data is generated. This roughly doubled the size of the polygons and thus also the number of returned objects. Still the complexity of the polygons is reduced significantly[2].

The left part of Figure 5.18 shows a logarithmic plot of the throughput. ACT with enabled support for textual predicates reaches a throughput of more than 20 million objects per second. It outperforms all its competitors, as well as the original ACT. As stated before, the rather cheap bloom filter and term containment checks avoid the time-consuming point-in-polygon checks.

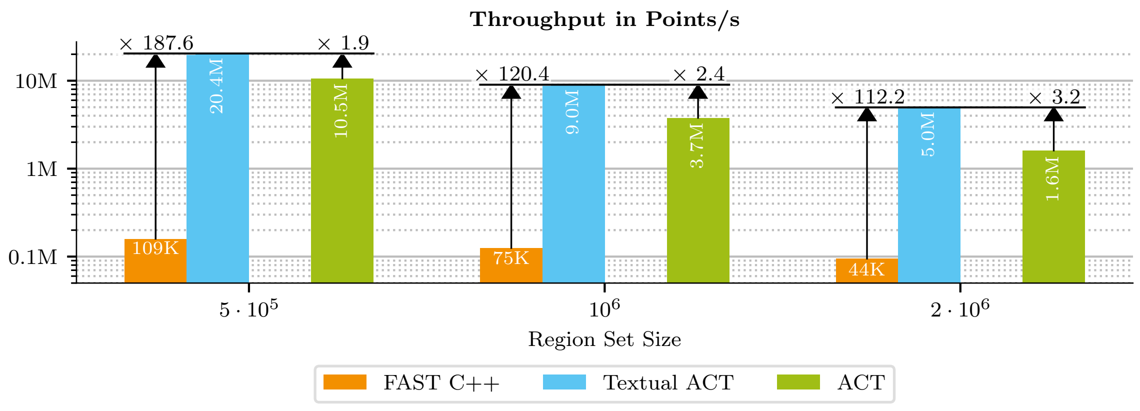


Figure 5.18: Performance Comparison of ACT and FAST C++

ACT is also an order of magnitude faster than FAST C++ and thus also FAST Java. Its single threaded run outperforms FAST by almost up to 200 times. This performance was possible

[2]The number of support points is reduced by 87%

because the workload matched the design parameters of ACT. It shows that the combination of an ACT with textual filtering has the potential to outperform all other competitors.

6 Conclusions

This thesis contributes to several areas of spatio-textual joins. It gives a short overview of techniques used in state-of-the-art spatio-textual and geospatial index structures. The Adaptive Keyword Index (AKI) is extracted from the efficient state-of-the-art index FAST. The throughput of AKI was further optimized achieving a speed-up of up to 3x by adding a keyword dictionary, small bit-filters for pruning and a reduced storage layout of the nodes. The keyword dictionary and the altered storage layout additionally reduce the memory consumption by up to 50%.

The throughput optimized spatio-textual Adaptive Cell Trie (ACT) is extended with textual support. We integrated the textual pruning structure inside ACT to avoid as many time-consuming operations, like point-in-polygon tests, as possible. Furthermore, the bit-filters also used in AKI are integrated into ACT to increase the locality and reduce cache misses.

The authors of FAST provided their workload for the evaluation. However, it did not feature very realistic spatio-textual join data. Therefore, we designed a dataset generator and used it with real-world point of interest data to generate three new datasets. These datasets resemble a more realistic workload.

These four datasets were used to profile AKI. The optimization steps are measured individually and in combination the data-structure has improved and is now more than three times faster than before. Since all textual indexes use hash tables to a high degree, several different hash table implementations were tested. Thereby, we managed to double the performance of AKI and the throughput of most of the competitors as well. Afterwards, AKI and its state-of-the-art competitors were compared with different data set sizes. AKI is on par with its fastest competitor. Unlike the competitors, it is robust against changes of the workload, making it the overall fastest index.

The FAST index was refactored and reimplemented in modern C++. This implementation is tested against the Java implementation to reproduce the original measurements. The C++ implementation provides a speed-up of up to 25% for the original workload. Because modern C++ features like constant functions were used, parallelization works easily and the algorithm scales almost linearly with the thread count.

ACT was tested for a spatially disjoint workload and outperforms FAST C++ by several orders of a magnitude.

Future Work Based on this thesis' foundations, spatio-textual joins can be made even more efficient and versatile. All optimizations which were introduced for stand-alone AKI can be merged back to the FAST index to increase both the throughput and the memory efficiency as well. The geospatial ACT optionally uses the AVX512 instruction set to fully use the capacity of the hardware. Those instructions are not used by the data structures for AKI and the textual part of ACT. However, these data structures are candidates to benefit from vectorization due to their design.

As of yet, FAST and textual ACT only feature one type of textual selection, namely subscription indexes. This may be expanded to allow for different types of textual queries like cross-sections or even similarity. For this purpose, different state-of-the-art text processing engines may be integrated to ACT.

While working on this thesis, an approach to perform geospatial joins on GPUs with some similarities to ACT has been presented. This work may be combined with the ray-tracing and quad-tree features of the recently presented Turing generation of Nvidia GPUs. These sparse grids would push limits of working with these workloads on GPUs.

In conclusion, this thesis uses novel approaches to lay the foundation for spatio-textual data processing on modern hardware. The presented optimizations to the AKI reach the same level as other state-of-the-art indexes. Spatio-textual ACT provides superior performance for disjoint workloads.

List of Figures

1.1 Mobile Phone Ownership in the U.S. 1
1.2 Various Yelp Usage Metrics 2
1.3 Comparison of FAST and ACT with their Respective Competitors 2
1.4 Schematic Example of Pollination in Germany 3
1.5 Schematic Example of Advertisement 4
1.6 Problem Defintion 6
1.7 Different Types of Spatial Index Structures 7
1.8 Different Types of Textual Index Structures 8

2.1 Cells at Different Levels Enumerated by S2 using the Hilbert Curve 11
2.2 Adaptive Ranked Inverted List Insert and Match 14
2.3 Ordered Keyword Trie Insert and Match 16
2.4 Adaptive Keyword Index Construction and Match 16
2.5 Schematic Structure of FAST 17
2.6 Spatial Sharing of Region Lists in FAST 18
2.7 Bloom Filters for Multi-Key Containment 19

3.1 Performance Measurements for AKI with and without Keyword Dictionary . . 24
3.2 Total Function Calls using Bloom Filter 25
3.3 Performance Improvements using Bloom Filter 25
3.4 Reducing Terms in AKI 27
3.5 Average Length of Term Lists 27
3.6 Benefits of Optimized Term Processing 28
3.7 Performance Increase due to Term Reduction 28
3.8 Performance Comparison with and without Simple Term Filtering 29
3.9 Performance Comparison of different Lookup Tables 30
3.10 Performance Comparison with Bloom Filtering 31
3.11 Subset Containment Call Reduction with Bloom Filtering 31
3.12 Performance of Bloom Filtered AKI compared with Bloom Filtered Sorted List 32
3.13 Performance of ACT with and without AKI for cells with more than 100 Regions attached 32
3.14 Statistics of intermediate building data structures 33
3.15 Comparison of Cell Replication for AKI using Trie or Hash Table 34

4.1 Generation of Rectangles and Polygons around Support Point 35
4.2 Irregular domain-filling Polygons using joined Voronoi Cells 36
4.3 Zipfian Distributed TV Subtitles Data 37
4.4 Cities in Foursquare Dataset 38
4.5 K-Means Clustering on Yelp Data 39

4.6 Difference between Normal and Robust Covariance 40
4.7 *k*-Core of Keyword Graph from Stuttgart 41

5.1 Spatial Comparison between Naturally Inspired Regions and Objects 44
5.2 Textual Comparison between Naturally Inspired Regions and Objects 44
5.3 Spatial Comparison between Yelp Regions and Objects 45
5.4 Textual Comparison between Yelp Regions and Objects 45
5.5 Spatial Comparison between Twitter Regions and Objects 46
5.6 Textual Comparison between Twitter Regions and Objects 46
5.7 Correlation between Initial Character and Occurrence Compared between Regions and Objects 47
5.8 Lookup Performance of Hash Table Implementations 48
5.9 Performance Difference with different Hash Table Implementations 49
5.10 Performance Comparison of Subscription Indexes 50
5.11 Performance Comparison of AKI Competitors with different Hash Table Implementations 50
5.12 Performance Comparison under varying Frequent Term Threshold 52
5.13 Performance Comparison under varying Spatial Range 52
5.14 Performance Comparison under varying Number of Terms 53
5.15 Performance Comparison under varying Number of Regions 53
5.16 Performance when Increasing Thread Count 54
5.17 Performance when varying Selectivity 54
5.18 Performance Comparison of ACT and FAST C++ 55

List of Tables

1.1 Notations used throug the Thesis 5

2.1 Textual Dataset for Index Examples 13

3.1 Comparison of Cell Count for Uniform and Clustered Yelp Datasets 33

4.1 Number of Businesses by Metropolitan Area 40
4.2 Top 20 Categories in Stuttgart 41

5.1 Key Figures of the used Workloads 47
5.2 Overview over FAST Parameters used in Experimental Evaluation 51

Bibliography

[Bec+90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles.” In: *SIGMOD Rec.* 19.2 (May 1990), pp. 322–331. ISSN: 0163-5808. DOI: 10.1145/93605.98741.

[Blo70] B. H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors.” In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.

[Che+13] L. Chen, G. Cong, C. S. Jensen, and D. Wu. “Spatial Keyword Query Processing: An Experimental Evaluation.” In: *Proc. VLDB Endow.* 6.3 (Jan. 2013), pp. 217–228. ISSN: 2150-8097. DOI: 10.14778/2535569.2448955.

[Chr+11] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. “Text vs. Space: Efficient Geo-search Query Processing.” In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM ’11. Glasgow, Scotland, UK: ACM, 2011, pp. 423–432. ISBN: 978-1-4503-0717-8. DOI: 10.1145/2063576.2063641.

[CSM06] Y.-Y. Chen, T. Suel, and A. Markowetz. “Efficient Query Processing in Geographic Web Search Engines.” In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: ACM, 2006, pp. 277–288. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142505.

[Dav16] H. Dave. *Frequency Word List*. https://github.com/hermitdave/FrequencyWords. [Online; accessed October 24, 2018]. Aug. 2016.

[FHR08] I. D. Felipe, V. Hristidis, and N. Rishe. “Keyword Search on Spatial Databases.” In: *2008 IEEE 24th International Conference on Data Engineering*. Apr. 2008, pp. 656–665. DOI: 10.1109/ICDE.2008.4497474.

[Fou18] Foursquare Labs, Inc. *Foursquare: About*. https://foursquare.com/about/. [Online; accessed October 24, 2018]. 2018.

[Goo16] Google. *sparsehash – C++ associative containers*. https://github.com/sparsehash/sparsehash. [Online; accessed October 24, 2018]. July 2016.

[Gut84] A. Guttman. “R-trees: A Dynamic Index Structure for Spatial Searching.” In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266.

[Hme+12] Z. Hmedeh, H. Kourdounakis, V. Christophides, C. du Mouza, M. Scholl, and N. Travers. “Subscription Indexes for Web Syndication Systems.” In: *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT ’12. Berlin, Germany: ACM, 2012, pp. 312–323. ISBN: 978-1-4503-0790-1. DOI: 10.1145/2247596.2247634.

[HSS08] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX." In: *Proceedings of the 7th Python in Science Conference*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. Pasadena, CA USA, 2008, pp. 11–15.

[J+01] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. `http://www.scipy.org/`. [Online; accessed October 24, 2018]. 2001–.

[Kip+18] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. "Adaptive Geospatial Joins for Modern Hardware." In: *arXiv preprint* (2018). arXiv: `1802.09488`.

[Kip17] A. Kipf. "Scalable Geospatial Data Processing." In: *Proposal* (Nov. 2017).

[KSL10] A. Khodaei, C. Shahabi, and C. Li. "Hybrid Indexing and Seamless Ranking of Spatial and Textual Features of Web Documents." In: *Database and Expert Systems Applications*. Ed. by P. G. Bringas, A. Hameurlain, and G. Quirchmayr. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 450–466. ISBN: 978-3-642-15364-8.

[Li+13] G. Li, Y. Wang, T. Wang, and J. Feng. "Location-aware Publish/Subscribe." In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, 2013, pp. 802–810. ISBN: 978-1-4503-2174-7. DOI: `10.1145/2487575.2487617`.

[LT16] P. Lison and J. Tiedemann. "Opensubtitles2016: Extracting large parallel corpora from movie and tv subtitles." In: European Language Resources Association, 2016, pp. 923–929. ISBN: 978-2-9517408-9-1.

[MAA17] A. R. Mahmood, W. G. Aref, and A. M. Aly. "FAST: Frequency-Aware Spatio-Textual Indexing for In-Memory Continuous Filter Query Processing." In: *arXiv preprint* (2017). arXiv: `1709.02529`.

[Mah+15] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah. "Tornado: A Distributed Spatio-textual Stream Processing System." In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 2020–2023. ISSN: 2150-8097. DOI: `10.14778/2824032.2824126`.

[Mah+17] A. R. Mahmood, A. Daghistani, A. M. Aly, W. G. Aref, M. Tang, S. M. Basalamah, and S. Prabhakar. "Adaptive Processing of Spatial-Keyword Data Over a Distributed Streaming Cluster." In: *arXiv preprint* (2017). arXiv: `1709.02533`.

[Mah18] A. R. Mahmood. "ACM SIGSPATIAL: G: Scalable Query Processing In Spatio-Textual Data Management Systems." In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL'18. Seattle, Washington: ACM, 2018.

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Pew18] Pew Research Center. *Mobile Fact Sheet*. `http://www.pewinternet.org/fact-sheet/mobile/`. [Online; accessed October 24, 2018]. Jan. 2018.

[Pow98] D. M. W. Powers. "Applications and Explanations of Zipf's Law." In: *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*. NeMLaP3/CoNLL '98. Sydney, Australia: Association for Computational Linguistics, 1998, pp. 151–160. ISBN: 0-7258-0634-6.

[Roc+10] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. "Efficient Processing of Top-k Spatial Preference Queries." In: *Proc. VLDB Endow.* 4.2 (Nov. 2010), pp. 93–104. ISSN: 2150-8097. DOI: `10.14778/1921071.1921076`.

[Ska18a] M. Skarupke. *Fibonacci Hashing: The Optimization that the World Forgot (or: a Better Alternative to Integer Modulo)*. `https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/`. [Online; accessed October 24, 2018]. June 2018.

[Ska18b] M. Skarupke. *flat_hash_map – A very fast hashtable*. `https://github.com/skarupke/flat_hash_map`. [Online; accessed October 24, 2018]. July 2018.

[Tes16] Tessil. *Benchmark of major hash maps implementations*. `https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html`. [Online; accessed October 24, 2018]. Aug. 2016.

[Vai+05] 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*. Ed. by C. Bauzer Medeiros, M. J. Egenhofer, and E. Bertino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 218–235. ISBN: 978-3-540-31904-7.

[Vea+18] E. Veach, J. Rosenstock, E. Engle, R. Snedegar, and J. Basch. *S2 Geometry Library*. `https://s2geometry.io`. [Online; accessed October 24, 2018]. Jan. 2018.

[Wan+15] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. "AP-Tree: Efficiently support continuous spatial-keyword queries over stream." In: *2015 IEEE 31st International Conference on Data Engineering*. Apr. 2015, pp. 1107–1118. DOI: `10.1109/ICDE.2015.7113360`.

[Wu+12] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. "Joint Top-K Spatial Keyword Query Processing." In: *IEEE Transactions on Knowledge and Data Engineering* 24.10 (Oct. 2012), pp. 1889–1903. ISSN: 1041-4347. DOI: `10.1109/TKDE.2011.172`.

[Yel18a] Yelp Inc. *An Introduction to Yelp Metrics as of June 30, 2018*. `https://www.yelp.com/factsheet`. [Online; accessed October 24, 2018]. June 2018.

[Yel18b] Yelp Inc. *Yelp Open Dataset: An all-purpose dataset for learning*. `https://www.yelp.com/dataset`. [Online; accessed October 24, 2018]. Aug. 2018.

[YZQ16] D. Yang, D. Zhang, and B. Qu. "Participatory Cultural Mapping Based on Collective Behavior Data in Location-Based Social Networks." In: *ACM Trans. Intell. Syst. Technol.* 7.3 (Jan. 2016), 30:1–30:23. ISSN: 2157-6904. DOI: `10.1145/2814575`.

[Zac+17] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freiref. "GPU Rasterization for Real-time Spatial Aggregation over Arbitrary Polygons." In: *Proc. VLDB Endow.* 11.3 (Nov. 2017), pp. 352–365. ISSN: 2150-8097. DOI: `10.14778/3157794.3157803`.

[Zho+05] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. “Hybrid Index Structures for Location-based Web Search.” In: *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. CIKM ’05. Bremen, Germany: ACM, 2005, pp. 155–162. ISBN: 1-59593-140-6. DOI: `10.1145/1099554.1099584`.

[ZM06] J. Zobel and A. Moffat. “Inverted Files for Text Search Engines.” In: *ACM Comput. Surv.* 38.2 (July 2006). ISSN: 0360-0300. DOI: `10.1145/1132956.1132959`.