UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CÍCERO AUGUSTO DE LARA PAHINS

Real-Time Exploration and Analysis of Big Data

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. PhD. João Comba

Porto Alegre
June 2018

CIP — CATALOGING-IN-PUBLICATION

Pahins, Cícero Augusto de Lara

Real-Time Exploration
and Analysis of Big Data / Cícero Augusto de Lara Pahins. – Porto Alegre: PPGC da UFRGS, 2018.

174 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: João Comba.

1. Data structures. 2. Big data. 3. Real-time. 4. Spatiotemporal. I. Comba, João. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof^{a}. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof^{a}. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Dedicated to my family and wife.”

ACKNOWLEDGEMENT

I thank my family for believing in me during the long years of my academic formation, always giving me support to continue the studies and fulfill my dreams. A very special and warm thank you to my parents. I owe it all to you.

I am grateful to my wife Fernanda, which has a very important role in the conclusion of this thesis, as well as in all aspects of my life. Thank you for always being by my side, no matter what situation.

Last but not the least, I also would like to express my gratitude to my advisor Prof. João Comba for the support during the entire period of my PhD, either through technical guidance or in the writing of the papers, as well as for the valuable words of advice to my life in general.

Thank you all.

ABSTRACT

This thesis consists of developing methods to enable the real-time exploration and analysis of big data. The solutions must be both memory and run-time efficient, as well as take into consideration the (i) scale of data, (ii) different forms of data, (iii) analysis of streaming data and (iv) uncertainty of data. Relational databases, or statistical packages, have difficulty to handle large multidimensional datasets. Naive solutions can take prohibitively large amounts of memory or time to answer as the number of dimensions increases. The interactive visualization of large datasets follows two main strategies: sampling and pre-computation. One limitation of the sampling strategy is the non-trivial extraction of random samples of large datasets, and naïve sampling strategies can generate biased results. This research mainly focuses on pre-computation strategies, which relies on the idea of computing aggregations over several dimensions. The core bottleneck of this strategy is the large memory footprint that is common to data structures used to accelerate data queries, e.g., data cube methods. Nevertheless, the real-time exploration and analysis of big data are one of the primary desires of visualization practitioners and data scientists. This thesis discusses the problem and presents the author's contributions.

Keywords: Data structures. big data. real-time. spatiotemporal.

Exploração e Análise de *Big Data* em Tempo Real

RESUMO

Esta tese consiste em desenvolver métodos para permitir a exploração e análise em tempo real de big data. As soluções devem ser eficientes em termos de memória e de tempo de execução, bem como levar em consideração a (i) escala de dados, (ii) diferentes formas de dados, (iii) análise de dados de streaming e (iv) incerteza de dados. Bancos de dados relacionais, ou pacotes estatísticos, têm dificuldade em lidar com grandes conjuntos de dados multidimensionais. As soluções ingênuas podem consumir quantidades proibitivamente grandes de memória ou tempo para responder à medida que o número de dimensões aumenta. A visualização interativa de grandes conjuntos de dados segue duas estratégias principais: amostragem e pré-computação. Uma limitação da estratégia de amostragem é a extração não trivial de amostras aleatórias de grandes conjuntos de dados, e estratégias de amostragem ingênuas podem gerar resultados tendenciosos. Esta pesquisa foca principalmente em estratégias de pré-computação, as quais se baseiam na idéia de pré-computar agregações. O principal gargalo dessa estratégia é a grande quantidade de memória comum às estruturas de dados usadas para acelerar consultas de dados, por exemplo, métodos de cubo de dados. Mesmo assim, a exploração e a análise em tempo real de big data são um dos principais desejos. de praticantes de visualização e cientistas de dados. Esta tese discute o problema e apresenta as contribuições do autor.

Palavras-chave: Estrutura de dados, big data, tempo real, spatiotemporal.

LIST OF ABBREVIATIONS AND ACRONYMS

HC	Hashedcubes
PMQ	Packed-Memory Quadtree
QDS	Quantile Data Structure
SC	SimilarityCubes
qnt	Quantile
KS	Kolmogorov-Smirnov

LIST OF FIGURES

Figure 2.1 HashedCube building steps for ten points [o0, ..., o9]. The process is described in Section 2.3. 31
Figure 2.2 An illustration of how to interpret pivot hierarchy. HashedCubes reassembles tree hierarchy. 33
Figure 2.3 Spatial dimension indexing scheme. A period of time is represented by a timestamped list of pivots. 35
Figure 2.4 HashedCubes enables real-time exploratory interactive systems using a wide range of visual encodings, such as heat maps and choropleth maps, histograms, binned scatter plots, parallel sets, and others. They support brushing & linking across any dimension. The images show the heatmap and charts associated with two given queries. 37
Figure 2.5 HashedCubes temporal and spatial indexing schemes allow the exploration and analysis of check-ins as global trends, as well as geographically restrict events. In October 2008 Brightkite Iphone app goes live. In October 2009 Brightkite 2.0 was released. 38
Figure 2.6 By indexing Day of Week and Hour of Day as categorical dimensions of the Brightkite dataset, HashedCubes enables to highlight user interactions across specific geographical regions. There is a significant usage difference between US and Japan....38
Figure 2.7 (a) Growth of the number of pivots when inserting objects into HashedCubes. Notice the *key saturation* effect. (b) HashedCubes memory usage is directly proportional to the number of pivots. From top to bottom: Flights, Brightkite and Gowalla datasets. 39

Figure 3.1 Hashedcubes accelerates queries used in a wide range of interactive exploratory visualizations, such as heatmaps, time series plots, histograms and binned scatterplots, and supports brushing and linking across spatial, categorical and temporal dimensions. In this figure, we show some example visualizations backed by Hashedcubes. The left image shows 210.6 million tweets from November 2011 to June 2012, highlighting the activity during Superbowl XLVI. The central image shows 24.5 million pick-up locations of NYC green taxis rides from January 2014 to June 2015. On the right, the visualizations show different aspects of 4.5 million Brightkite check-ins, a social network. Hashedcubes balances low memory usage, fast running times, and simple implementation; it allows interactive exploration of datasets that previously either required a prohibitive amount of memory or uncomfortably large latencies. 44
Figure 3.2 Overall summary for building Hashedcubes. **(a)** Input dataset of points [p_0,...,p_9] under a spatial-categorical-temporal schema. The complete process is described in Section 3.4. **(b)** Step-by-step illustration of the process for building arrays of sorted partitions, as explained in Section 3.4.2. **(c)** Data is loaded (in any order) into a sequential memory and each record is associated with an index (rectangle in orange). 46
Figure 3.3 A comparison between the computation of Nanocubes and Hashedcubes. Note that Nanocubes pre-compute more aggregations, which tends to lead to lower query times but larger memory consumption. Hashedcubes, in contrast, uses a sparser set of preaggregations in its query execution engine....50

Figure 3.4 Multiple spatial dimensions. In this example one quadtree is created for each of the two spatial dimensions, red and blue. The quadtrees are used alternately in Hashedcubes to partition the data....52

Figure 3.5 Temporal dimension indexing. A period of time is represented by a dense list of timestamped pivots. Each black circle represents a record that has been tagged to a specific bin....53

Figure 3.6 Visual exploration of the twitter dataset during Super Bowl 2012. In addition to enabling real-time exploration using a wide range of visual encodings, with support to brushing & linking in any dimension, Hashedcubes allows the access to the text of tweets from an external SQL server....55

Figure 3.7 **(a)** Hashedcubes memory usage growth while inserting SPLOM dataset elements. Notice the *key saturation* effect. **(b)** and **(c)** compare Hashedcubes construction time and memory usage to Nanocubes....60

Figure 3.8 Cumulative percentages of query latency from real-world scenarios. The vast majority of queries are answered within the real-time budget (<40ms *or* >25fps) for different schemas and datasets....62

Figure 3.9 Hashedcubes supports recovering the original data by using a linking structure. Pivots represent the values from the SQL index, which allows to efficiently match all rows of a given query. Hashedcubes can be built directly from a SQL database or from an intermediary format....64

Figure 3.10 Hashedcubes different heatmap visualizations showcase. Notice the leaf size variation from 32 to 8 by looking into the highlighted regions. It impacts running time, memory usage and visual accuracy. **(a)** allows to identify truncated pivot occurrences by representing them as rectangles. Color is a factor of area and occupancy. **(b)** and **(c)** use circles to represent the center of an aggregated region (i.e., quadtree bounding box)....65

Figure 3.11 Los Angeles (United States) city view of detailed Brightkite heatmaps from recent data cube visualization proposals. Apart from the use of different colormaps across Hashedcubes, Nanocubes and imMens, what produces a slightly dissimilar visual appearance, Hashedcubes pivot concept enables a high visual accuracy along with reduced memory consumption when compared against other data cube visualization proposals. Notice that Hashedcubes matches Nanocubes visual representation, even though the latter does not experience leaf-size trade-offs....65

Figure 4.1 Analyzing the distribution of flight arrival delays for U.S. airports using QDS. We observe two maps showing the probability of flights being late for January and December 2014. Airports are colored using a divergent color scale representing the cumulative distribution function of the arrival delays at the value 0. We assign red color shades to airports with a higher probability of having late arriving flights and blue shades for airports in which flights are more likely to be early. Notice how the trend changes from more likely delayed flights on the Northeastern airports in January to Southwestern ones (particularly in California) in December. The pattern of delay in January 2014 is due to the snowstorms that pounded the Northeast of the U.S. in January. The Western delay pattern in December is due to the so-called “California’s storm of the decade" that affected the region in the middle of December 2014. The temporal band plots on the bottom show the evolution of the arrival delay quantiles (0.1,0.25,0.5,0.75,0.9) for both the JFK (left) and SFO airport (right). Dates with a substantial increase in the median arrival delays (black line) are the peaks of these events (e.g., January 4 on the left and December 15 and 19 on the right). 69

Figure 4.2 Gaussian distributions are the most common approach of modeling data for analysis and visualization. While this method has theoretical advantages, real-world data is rarely normally distributed. As we observe in (a)-(c) modeling data with normal distributions (black curves) can introduce biases and hide essential features such as multimodality and skewness. As illustrated by the equi-depth histograms produced using p-digest in (d) (darker shades of blue represent higher data density) can efficiently describe the distributions of the other plots. 71

Figure 4.3 The t-digest sketch: (a) construction of a t-digest t_1. The *cdf* of the input dataset is represented by a set of weighted centroids. (b) Different quantile sketches $t1$ and $t2$ can be combined using the merge operation. (c) A quantile query, qnt(value), interpolates the centroid weights compared to the fraction of the total weight defined by the input value to compute the estimate of the result quantile. 73

Figure 4.4 Queries supported by QDS. Let F_M be the random field formed by merging quantile sketches of selected bins. (a) The quantile and cdf queries receive a parameter x and returns the result of the corresponding query for each quantile sketch. (b) The pipeline query: we use the result of a given query as a parameter to a second one using a right join process. In case the parameter has “missing” bins the result query can have undefined (purple X) values. 75

Figure 4.5 QDS indexing scheme and shared pointers. We use an example to compare the indexes of HC (a) and QDS (b). The input dataset has eight records, each with three dimensions. In HC, each dimension stores pivots in a pivot array that refers to intervals in the input dataset. In QDS, in addition to the pivot array for each dimension (primary pivot array), we keep a secondary pivot array for each element. In graphical terms, the primary array is displayed horizontally, while the secondary array is displayed vertically. Searching for values equal to F in QDS can be simply done by following vertically the secondary pivot array associated with F in dimension 2. The number of pivots stored in the QDS is not larger than in HC. Each pivot has an additional payload (marked with *) that can store quantile information. (c) Pivot arrays tend to have duplicate information across dimensions. To save memory, QDS used shared pointers to compact shared pivot and payload information. 76

Figure 4.6 The QDS query algorithm demonstrated using the dataset of Fig. 4.5. The input query has constraints in all dimensions. In the *selection* step, the pivot array of each dimension is processed to check the pivots that satisfy the query for that dimension. In the *intersection* step, we compute in sequence the intersection of the results of previous steps. The *aggregation* step compacts the results of the previous step. 79

Figure 4.7 Quantile heatmaps of taxi trip fares in NYC during the month of October 2014 based on their pick-up locations. The mean based heatmap (a) conveys high prices similar to the third quartile map(d). The median heatmap shows lower fares (c) indicating the robustness with respect to outliers. The first quartile map (b) indicates mostly lower values except on regions close to the Queens–Midtown Expressway near the high traffic region of Queens—Midtown Tunnel's toll station (right dashed box). 80

Figure 4.8 Box plot of flight arrival delays per carrier. The boxes are colored and sorted according to the probability of each distribution being below 15 minutes (red line), which represents the proportion of on-time flights. 83

Figure 4.9 Daily arrival delay outlierness in 2017 for Delta JetBlue and Southwest airlines. Delta had an abnormal first week of April due to severe weather in its hub city Atlanta. Similarly, weather events created abnormal arrival times for JetBlue and Southwest in May and August respectively. January 29th is another odd day: a computer outage grounded all Delta's flights. The news on the side corroborate the unexpected events found. 84

Figure 4.10 Exploration of NYC yellow taxi trips in October 2014. (a) Outlierness coefficients with respect to total fare vary widely during the month with peaks on days 10, 12, 17, 23, 24 and 31. This last one being the highest. (b) Analyzing how the outlierness vary over the day 31 we see that the day got more "unusual" with the highest values on the period starting at 7 PM. (c) The heatmap resulting from the pipeline query in this period we observe from left to right that trips are more expensive than normal in the Greenwich Village Region. Zooming in we see that a portion of the streets (purple) that unusually did not have any trips. This corresponds to the area where the annual Greenwich Village Halloween parade happened. 86

Figure 4.11 Performance comparison of QDS, HC and database alternatives computing count queries. QDS novel index successfully avoid HC corner cases and offers a query latency that typically lies below 10ms in various datasets. 89

Figure 4.12 Evaluation of p-digest's quantile estimation with respect to (a) pivot size, (b) number of merge operations and (c) queried quantile. 91

Figure 4.13 Comparison of $qnt(0.5)$ computation on a synthetic dataset using QDS, QDS without p-digest and database alternatives. As shown, QDS can provide quantile queries at interactive rates. 92

Figure 5.1 COVIZ architecture. 96

Figure 5.2 An instance of QDS indexing scheme for eight records and three dimensions. QDS stores at each pivot (marked with an asterisk) a payload that contains the representation of a distribution function. 98

Figure 5.3 Tasks in COVIZ: cohort formation (A-B), cohort comparison (C-D), sensemaking with pollution data (E-F). 100

Figure 6.1 Overview of the author's QDS data cube proposal. It employs a novel data cube indexing that reduce memory usage of previous methods and introduces p-digest (de Lara Pahins; Ferreira; Comba, 2019), an efficient quantile sketch that follows the theoretical requirements that an analytical method must implement to be integrated into the data cube infrastructure built by the author. 105
Figure 6.2 Overview of the process of computing the Fourier spectrum of a decision tree and vice-versa. Adapted from (KARGUPTA; PARK, 2004). A *proof-of-concept* implementation was proposed by the author of this thesis in Appendix C. 106
Figure 6.3 Illustration of the two-sample Kolmogorov–Smirnov test. The black arrow is the KS distance between two one-dimensional probability distributions. 107
Figure 6.4 p-digest enables the representation of a set of directions by modeling it's distribution into the range (0 to 2π). By using the KS distance, the clustering algorithm can measure the similarity between distributions that represent directions. 108
Figure 6.5 Collection of simulated scenes and datasets with varying properties. (a) Trajectories obtained on a four lane highway with traffic in both directions. (b) Trajectories obtained by visual tracking of vehicles from a highway. (c) Trajectories obtained from a four traffic intersection. (d) Trajectories obtained by tracking a laboratory. Adapted from (MORRIS; TRIVEDI, 2009). 109
Figure 6.6 Visualization of 8 clusters generated by QDS using Morris et al. dataset (MORRIS; TRIVEDI, 2009). 110
Figure 6.7 Overview of the *HURDAT2* dataset. The data consists of tracking information of all known tropical and subtropical cyclones from 1851 to 2017 (LANDSEA; FRANKLIN, 2013). 112
Figure 6.8 Visualization of 8 clusters generated by QDS using HURDAT2 dataset. Each hurricane path is clustered based on location, wind and pressure features. 113

Figure A.1 Z-curve algorithm explanation. 132
Figure A.2 Hilbert curve, second order example. 132
Figure A.3 Quantifying the neighborhood preservation of four datasets performed by Fiedler Vector (🟥), Z Order Curve (🟦) and Hilbert Curve (🟩), the result of this measure is impacted by the instances distribution along the dimension. The best method to use depends in how well the data is described by a Z curve or how many times a long jump is avoided. 133
Figure A.4 A summary of our proposed methodology for similarity queries in a high dimensional space. The main idea is to pre-process the high dimensional dataset to obtain a simplified but descriptive dataset in one-dimensional space to enable the computation of similarity queries in real-time 134
Figure A.5 An overview of our method. First we obtain a single value representation given by a multidimensional projection for every record in the dataset, so that we execute a sorting step that results in an array of sorted elements by similarity. 134

Figure A.6 Concrete example of the our approaches with the same three query points showing different results in every case. 135

Figure A.7 Exploring the Brightkite checking similarity distribution using five different colormaps that enables a visual differentiation of the similarity over a geographical map. 138

Figure A.8 Visual exploration of the New York City Civilian Complaints dataset using four different query points and range queries. 139

Figure A.9 Visualizing 210 million public geolocated Twitter showing our three approaches to retrieve the most similar elements against multiple points. 140

Figure B.1 A Twitter stream is consumed in real-time, indexed and stored in the Packed-Memory Quadtree. **(a)** : live heat-map displays tweets in the current time window. **(b)** : alerts indicate regions with high activity of Twitter posts at the moment. **(c)** : the interface allows to drill-down into any region and query the current data. **(d)** : the actual data can be retrieved from the Packed-Memory Quadtree to analyze the tweets in the region of interest. 144

Figure B.2 PMQ with 4 segments: ρ_l and τ_l are the minimum and maximum densities allowed at each level l of PMQ, d the actual window density. The numbers in circles count the valid element per window and are stored in the PMQ accounting array. 148

Figure B.3 PMQ: data structure overview 149

Figure B.4 The heatmap is updated dynamically as the stream of tweets is received. With an average insertion rate of $1000\ tweets/sec$ we show the heatmap when PMQ contains $100K$ (left) or $10M$ (right) elements. 151

Figure B.5 Heatmap zoom and range queries are used to explore the latest stream of tweets around the New York city area. The in-memory storage of PMQ provides fast access to the actual tweets' content allowing real-time user interaction even on large range queries (R = radius of the selected area). 152

Figure B.6 Allert detection: triggers configured by the user show alerts (yellow squares) on several cities with a high rate of tweet arrival during Super Bowl 2012. We select the Indianapolis region (where the game occurs) and filter tweets using a top-k query to retrieve the most relevant tweets in the area. 155

Figure B.7 Performance comparison of spatial data storage solutions. Top row: standard geospatial databases can not handle real-time insertions. Bottom row: in-memory containers based on dense or sparse (PMQ) vectors. 157

Figure B.8 Scalability insertion and scan operation. 158

Figure B.9 Steady data regime: deletions are performed periodically. For each test, we insert a dataset of 46 million elements. The maximum number of elements allowed is half of the dataset size (around 23 million elements), and removals are configured with different percentages of the maximum. 159

Figure B.10 PMQ performance at steady regime for different τ_h thresholds. 159

Figure B.11 PMQ average insertion time with a window of $6h$ and varying rates. Current Twitter insertion rate (6K tweets/s) can be processed under 7.5 ms. 162

Figure B.12 Range queries: PMQ speedup over the B-Tree and R-Tree. Each boxplot represents the speedup of throughput for each query instance. The average speedup is denoted by red crosses. PMQ is faster than the B-Tree in all cases. Compared to the R-Tree, PMQ has a speedup on 97% of queries tested. The cases where PMQ performs worst corresponds to queries returning a small number of elements compared to the dataset size. 163

Figure B.13 Examples of Range Queries. We define 8 different query widths. The B-Tree and PMQ use 10 levels of quadtree refinement for range queries. We choose this parameter experimentally to provide the best overall results. 164
Figure B.14 Top-k Queries: cumulative percentages of query latency for $K = 100$, $R = 30$ km and $T = 10000$ seconds. We compared the search performance of PMQ against the *Kite* framework. 165
Figure B.15 PMQ memory usage depends on density thresholds $2\rho_h < \tau_h$. `Boost C++` R-Tree has a bigger index overhead than `stx::btree`. 165

LIST OF TABLES

Table 2.1 Overall summary of the relevant information for building HashedCubes. *Not Measured. 34
Table 2.2 Subset of queries supported by HashedCubes. 36

Table 3.1 Subset of queries supported by Hashedcubes HTTP API. 53
Table 3.2 Overall summary of the relevant information for building Hashedcubes. 59

Table 4.1 Overall summary of the relevant information for building QDS. 87
Table 4.2 Compression results for different p-digest configurations. 93

Table A.1 Summarized differences between data structures addressed for this research. 129
Table A.2 Summary of the datasets used to evaluate Fiedler vector, Z-order and Hilbert curve projections. 132
Table A.3 Summary of the datasets considered for our results and the resource usage when indexing the categorical values for similarity queries. 137

Table B.1 Parameter λ set for the best relation of removal *RM* time and average *Avg* runnning time for each algorithm. 161
Table B.2 Comparison using same λ optimized for the B-Tree. 162
Table B.3 Insertion time of batches of 1K elements in a PMQ with different time-windows λ. The number of elements in the container varies from Elts_min to Elts_max. The Mean, 99% and Max times are in ms. 162
Table B.4 Memory usage summary 167

CONTENTS

1 INTRODUCTION 19
1.1 Background 20
1.2 Collection of Papers and Contributions 26
1.2.1 Real-Time Visual Exploration of Big Data 26
1.2.2 Similarity-based Visual Exploration of Multidimensional Datasets 27
1.2.3 Real-Time Visual Exploration and Analysis Based on Order Statistics 27
1.2.4 Visual Formation and Comparison of Patient Cohorts 27
1.2.5 Real-Time Visual Exploration of Streaming Big Data 28
2 HASHEDCUBES: A DATA STRUCTURE FOR REAL-TIME EXPLORATION OF LARGE MULTIDIMENSIONAL DATASETS 29
2.1 Abstract 29
2.2 Introduction 29
2.3 HashedCubes Concept 31
2.4 HashedCubes Construction 33
2.4.1 Spatial Dimensions 34
2.4.2 Categorical Dimensions 35
2.4.3 Temporal Dimensions 35
2.5 HashedCubes Queries 36
2.6 Experiments and Discussion 36
2.6.1 Result Analysis 38
2.7 Conclusions and Future Work 40
3 HASHEDCUBES: SIMPLE, LOW MEMORY, REAL-TIME VISUAL EXPLORATION OF BIG DATA 42
3.1 Abstract 42
3.2 Introduction 42
3.3 Related Work 43
3.4 Hashedcubes 46
3.4.1 Some Intuition 47
3.4.2 Construction Algorithm 49
3.4.3 Spatial Dimensions 51
3.4.4 Categorical Dimensions 52
3.4.5 Temporal Dimensions 53
3.4.6 Queries 54
3.5 Implementation 55
3.6 Datasets and Schemas 56
3.6.1 Location-Based Social Networks 56
3.6.2 Airline On-Time Performance 57
3.6.3 SPLOM 57
3.6.4 Twitter 58
3.6.5 NYC Yellow and Green Taxis 58
3.7 Performance Results 58
3.7.1 Memory Usage 59
3.7.2 Construction Time 61
3.7.3 Query Time 61
3.8 Discussion 62
3.9 Conclusions and Future Work 64

4 REAL-TIME EXPLORATION OF LARGE SPATIOTEMPORAL DATASETS BASED ON ORDER STATISTICS 67
4.1 Abstract 67
4.2 Introduction 67
4.3 Related Work 69
4.4 Background 71
4.5 The t-digest data sketch 72
4.6 Quantile Data Structure 74
4.6.1 Overview and Query Types 74
4.6.2 Internal Representation 75
4.6.2.1 Indexing Scheme 76
4.6.2.2 The p-digest data sketch 77
4.6.3 Query Algorithm 78
4.6.4 Implementation 79
4.7 Building Visualizations with QDS 80
4.7.1 Extending Usual Visualizations 80
4.7.2 Easing the Reading of Uncertainty Visualizations 82
4.7.3 Uncovering the Unexpected 82
4.8 Use Cases 85
4.8.1 Analyzing Flights Delays 85
4.8.2 Exploring Outlierness in Taxi Trip Records 87
4.9 Experimental Results 88
4.9.1 The QDS Index Experiments 88
4.9.2 The p-digest Sketch Experiments 89
4.10 Discussion 92
4.11 Conclusions and Future Work 93
5 VISUAL FORMATION AND COMPARISON OF PATIENT COHORTS 94
5.1 Abstract 94
5.2 Introduction 94
5.3 System Design 96
5.3.1 Datasets 96
5.3.2 Cohort formation 97
5.3.3 Cohort exploration 99
5.4 Interface 101
5.5 Demonstration Scenarios 102
6 CONCLUSIONS AND DISCUSSION 104
6.1 A Fourier Spectrum-Based Approach to Represent the Decision Tree Classifier and Regression Method 106
6.2 A Clustering Technique 107
REFERENCES 114
APPENDICES 126
APPENDIX A SIMILARITY-BASED VISUAL EXPLORATION OF VERY LARGE GEOREFERENCED MULTIDIMENSIONAL DATASETS 127
A.1 Abstract 127
A.2 Introduction 127
A.3 Related Work 129
A.4 Our approach 131
A.4.1 Projecting data to 1D 131
A.4.2 Indexing data 133
A.4.3 Querying data 135
A.4.4 Implementation details 135

A.5 Results **136**
A.6 Discussion and Limitation **139**
A.7 Conclusion **141**
APPENDIX B PACKED-MEMORY QUADTREE: A CACHE-OBLIVIOUS DATA STRUCTURE FOR VISUAL EXPLORATION OF STREAMING SPATIOTEMPORAL BIG DATA **142**
B.1 Abstract **142**
B.2 Introduction **142**
B.3 Related Work **144**
B.4 Packed-Memory Quadtree **147**
B.4.1 The PMQ Data Structure 147
B.4.2 Data Indexing 149
B.4.3 Dynamic Updates 149
B.4.4 Query Types 151
B.5 Implementation **153**
B.6 Examples of Visualization Analysis using PMQ **154**
B.6.1 Drill-down exploration of a Twitter stream 154
B.6.2 Allert detection of regions with high tweet rates 154
B.7 Performance Evaluation **155**
B.7.1 Evaluating Storage Solutions for Spatial Data 156
B.7.2 Evaluating Insertions 157
B.7.3 Evaluating Bulk Deletions 160
B.7.4 Evaluating the Rebalancing Procedure 161
B.7.5 Evaluating Range Queries 162
B.7.6 Evaluating Top-k Queries 165
B.7.7 Evaluating Memory Usage 166
B.7.8 Discussion of the Evaluation Results 167
B.8 Conclusion and Future Work **168**
B.9 Proof of the PMQ Amortized Cost **168**
APPENDIX C IMPLEMENTATION OF FOURIER SPECTRUM-BASED APPROACH TO REPRESENT DECISION TREES **170**

1 INTRODUCTION

Data acquisition has never been so broad, diverse, and accessible. Nowadays, almost every computer-based gadget offers a broad range of options to collect multidimensional data. Social networking, government data, application records, and many others generate immense amounts of raw data.

A fundamental problem in modern visual data analysis is how to build data exploration environments that support interactive exploration of large datasets. This problem has two opposing facets. From one side, the ever-growing complexity and size of datasets bring the need to provide complex navigation and visual summaries capabilities. On the other hand, human perception and cognition pose a challenge on how long the data handling and rendering loop can take. Even small delays on the scale of half a second can have a significant negative impact on the visual data exploration process (LIU; HEER, 2014b). Unfortunately, the ability to produce compelling visual summaries, interaction mechanisms and interfaces (ELMQVIST; FEKETE, 2010; FERREIRA; FISHER; KONIG, 2014) has surpassed our capabilities to create techniques that support real-time data processing for visualization (FEKETE et al., 2012). As a result, there are limitations on the analysis that one can hope to perform interactively.

Traditional tools such as relational databases, or statistical packages, have difficulty in handling large multidimensional datasets. Naive solutions can take prohibitively large amounts of memory or time to answer as the number of dimensions increases. Thus, specialized data structures to accelerate query time in these datasets are necessary.

Another aspect is that most approaches focus on large static datasets, but there is a growing interest in analyzing and visualizing data streams upon generation. Twitter is a typical example. The stream of tweets is continuous, and users want to be aware of the latest trends. This need is expected to grow with the Internet of things (IoT) and the massive deployment of sensors that generate large and heterogeneous data streams. Over the past years, several in-memory big-data management systems have appeared in academia and industry. In-memory databases systems avoid the overheads related to traditional I/O disk-based systems and have made it possible to perform interactive data-analysis over large amounts of data. A vast literature of systems and research strategies deals with different aspects, such as the limited storage size and a multi-level memory-hierarchy of caches (ZHANG et al., 2015). Maintaining the right data layout that favors the locality of accesses is a determinant factor for the performance of in-memory processing systems.

Stream processing engines commonly support the concept of *window*, which collects the latest events without a specific data organization. It is possible to trigger the analysis upon the occurrence of a given criterion (time, volume, specific event occurrence). After a window is updated, the system shifts the processing to the next batch of events. There is a need to go one step further to keep a live window continuously updated while having a fine grain data replacement policy to control the memory footprint. The challenge is the design of dynamic data structures to absorb high rate data streams, stash away the oldest data to stay in the allowed memory budget while enabling fast queries executions to update visual representations.

One limitation of current static and streaming solutions is the fact that most of the time they do not take into account the inherent *distribution uncertainty*: datasets with equal mean and covariance, but with entirely different underlying distributions. Examples of this issue can be seen in the classical Anscombe's Quartet datasets and the work of Matejka et al. (MATEJKA; FITZMAURICE, 2017). The state-of-the-art method Gaussian Cubes proposed by Wang et al. (Wang et al., 2017) supports interactive data modeling by describing the data distribution using parametric Gaussian distributions. Unfortunately, this approach has two drawbacks. First, it relies on non-robust statistics (mean and covariances), i.e., they can be easily affected by outliers. Second, and most importantly, one can not assume real-world data to be normal. Despite visualizations based on averages (or accompanied by some visual representation of variance) such as heatmaps, line plots and histograms are ubiquitous.

1.1 Background

In this section, we review related research on different aspects that play an important part in this thesis.

Interactive Visualization using Datacubes. The interactive visualization of large datasets follows two main strategies: sampling and pre-computation. The sampling strategy relies on online aggregation, *i.e.*, use progressively increasing samples of a population to approximate the result of a given query (FISHER et al., 2012a; MORITZ et al., 2017). In this scenario, users face evolving visualizations that indicate current estimates and, possibly, the uncertainty inherent to the estimation process. This estimation uncertainty brings extra complexity to decision making. While sophisticated interaction tools have been proposed to assist in data exploration based on sampling (FERREIRA; FISHER;

KONIG, 2014; MORITZ et al., 2017), the problem of dissociating estimation uncertainty from data uncertainty is still unexplored in visualization. On the other hand, the pre-computation strategy relies on the idea of computing aggregations over several dimensions following the datacube concept (GRAY et al., 1997a). The seminal paper of Gray et al. laid the foundation for many other methods (LIU; JIANG; HEER, 2013; Lins; Klosowski; Scheidegger, 2013; CAO et al., 2015). A data cube can be seen as a hierarchical aggregation of all data dimensions in an n-dimensional lattice. Its main disadvantage is its memory consumption, which becomes impractical as the number of dimensions increases. To address this problem, some approaches describe ways to compress data cubes, such as Dwarf (SISMANIS et al., 2002), Immens (LIU; JIANG; HEER, 2013) and Nanocubes (Lins; Klosowski; Scheidegger, 2013), or build on distributed databases to cope with scale requirements (KAMAT et al., 2014).

The imMens approach combines data reduction, multivariate data tiles, and parallel query processing (using a GPU) to minimize both data cube memory usage and query latency. Its multivariate data tile methods are based on the observation that for any pair of 1D or 2D binned plots, the maximum number of dimensions needed to support brushing and linking is four. Thus, an n-dimensional data cube can be decomposed into a collection of smaller 3- or 4-dimensional projections. Furthermore, these decomposed data cubes are segmented into multivariate tiles, like the ones used by Google Maps. On the other hand, imMens lacks support for compound brushing in more than four dimensions. Nanocubes is a compact variation of a data cube that can handle a large number of dimensions. It defines a search key that is used to combine aggregations of independent dimensions at varying levels of detail and to maximize shared links across the data structure. Recent systems such as TopKube (Miranda et al., 2018), SwiftTuna (JO et al., 2017), Gaussian Cubes (Wang et al., 2017) and Sesame (KAMAT; NANDI, 2018) extend ordinary datacubes to perform more complex analysis in real-time while respecting reasonable memory constraints. SwiftTuna incorporates frequency histograms and dot plots to reveal relationships among dimensions of the data. TopKube and Gaussian Cubes store different payloads in the cells of the cubes to support tasks such as ranking and modeling of data cube slices. Both Gaussian Cubes and Sesame incorporate uncertainty by modeling the data with parametric Gaussian distributions (mean and covariances). This approach is sensitive to outliers and may not only introduce bias but also hide the essential features of the data.

VisReduce (IM; VILLEGAS; MCGUFFIN, 2013) is an approach to data aggregation which computes visualization results in a distributed fashion. It uses a modified *MapReduce* (DEAN; GHEMAWAT, 2004) algorithm and data compression. Its main drawback is that interaction operations require on-demand aggregations. Thus, the final result is obtained only after the costly transfer over the network of partial and final aggregations. As a rule of thumb, on-demand computation is problematic for visual analysis because of latency. As Liu and Heer describe (LIU; HEER, 2014a), latencies of as little as half a second can affect the overall quality of an analyst's data exploration process. A popular alternative to hide latency is to use sampling and report uncertainty estimates as soon as they are available (FISHER et al., 2012b). Similarly, Stolper et al. describe a general framework for a progressive approach for visual analytics (STOLPER; PERER; GOTZ, 2014). The need for low latency in large databases is a popular theme in the literature (ASSENT et al., 2008; SHIEH; KEOGH, 2008; CAMERRA et al., 2010). *BlinkDB* (AGARWAL et al., 2013b) builds a carefully-constructed stratified sample of the dataset, which allows interactive latencies in approximate queries over multiple terabytes of data. In essence, *BlinkDB* provides infrastructure such that Hellerstein et al.'s online aggregation has fast convergence properties (HELLERSTEIN; HAAS; WANG, 1997). *ScalarR* improves performance by manipulating physical query plans and computing a dynamic reduction of query sets based on screen resolution (BATTLE; STONEBRAKER; CHANG, 2013); it is an early, central example of explicitly taking peculiarities of a visualization setup in a DB account. *3W* is a search framework for geo-temporal stamped documents that allows fast searches over spatial and text dimensions (NEPOMNYACHIY et al., 2014). *Forecache* (BATTLE; CHANG; STON, 2015) improves performance by predicting user actions ahead of the actual queries being issued. However, most of these solutions are not designed with visual exploration in mind.

The most recent trend in research at the intersection of data management and visualizations is the explicit acknowledgment of the human perceptual system. Wu et al. suggest that database engines should explicitly optimize for perceptual constraints, by for example, including the visual specification into the physical query planning process (WU; BATTLE; MADDEN, 2014). Jugel et al. offer a technique that is one such example: the query algorithms described there return approximate results which nevertheless rasterize to the same image as the exact query result would (JUGEL et al., 2014; JUGEL et al., 2016); ScalarR (BATTLE; STONEBRAKER; CHANG, 2013) is another example.

Uncertainty in Visual Analytics. Several survey papers summarize the state-of-art of uncertainty visualization (JOHNSON; SANDERSON, 2003; POTTER; ROSEN; JOHN-

SON, 2012; SACHA et al., 2016; KINKELDEY et al., 2017). Johnson and Sanderson (JOHNSON; SANDERSON, 2003) raised the issue of having a formal framework that incorporates errors and uncertainty information into visualization algorithms. This questioning for handling uncertainty data led to follow-up work that aimed at understanding the issues that arise in the visualization pipeline, as well as proposing new visualization algorithms. Several papers claim the importance of incorporating essential statistics into visualization pipelines (POTTER; GERBER; ANDERSON, 2013; MACIEJEWSKI et al., 2013; QUARTERONI, 2018). Statistical uncertainty can be used to identify events or anomaly situations, which is a powerful tool for visual analytics. Maciejewski et al. (MACIEJEWSKI et al., 2010) combine visual exploration with modeling strategies to find abnormal spatiotemporal hotspots. Also, Wilkinson et al. (WILKINSON, 2018) use a statistical algorithm for detecting multidimensional outliers. Other approaches such as computational topology (DORAISWAMY et al., 2014) and graph wavelet theory (VALDIVIA et al., 2015) have also been used in visual analytics systems.

Probability Theory and Data Sketches. We briefly discuss the background of probability theory and data sketches, and refer to Rosenthal (ROSENTHAL, 2006) and Cormode et al. (CORMODE et al., 2012) for a detailed description. We start with the concepts of distribution of a random variable and quantiles. We define the cumulative distribution function (*cdf*) of a random variable X by $F_X(t) = Pr(X \leq t)$. Quantiles are landmark values of a given *cdf* that define specific points where F_X has accumulated a fraction of its total probability. For example, a value t is the q^{th} quantile of F_x if $F_X(t) = q$. Intuitively, one can obtain the value of the q^{th} quantile by $F_X^{-1}(q)$ by simply inverting the *cdf*. In this presentation, the focus is on the intuition and overlook the fact that *cdf*'s are not necessarily invertible. We define the first (q_1), second (q_2) and third (q_3) *quartiles* as the quantiles that divide the density in four equal parts, *i.e.*, 0.25^{th}, 0.5^{th} and 0.75^{th} respectively. We define a *random field* as a function F_M that associates to each point in a spatial domain (*e.g.,* geographical coordinates) a random variable.

Unlike moment statistics, such as average and variance, quantiles are robust to the presence of outliers (WILKINSON, 2018). However, it is not possible to combine quantiles of different datasets (e.g., *cdfs*) without processing the input datasets entirely. This limits the use of quantiles in scenarios that require hierarchical/dynamic aggregation such as datacubes. An alternative is to use approximation schemes called *quantile sketches* (PHILLIPS, 2016). A data sketch is "a data structure that can be easily updated with new or modified data and supports a set of queries whose results approximate queries on the full dataset" (PHILLIPS, 2016). Quantile sketches are data sketches that support queries of quantile and *cdf* estimation, in particular for data streaming applications. Methods vary in memory usage and approximation performance, leading to two groups of methods. The first one has sketches that have proven approximation bounds such as the proposals of Shrivastava et al. (SHRIVASTAVA et al., 2004), Agarwal et al.(AGARWAL et al., 2013a), Karnin et al. (KARNIN; LANG; LIBERTY, 2016) and Felber and Ostrovsky (FELBER; OSTROVSKY, 2017). Such methods have performance requirements which incur in complex algorithms that use large amounts of memory in practice (see discussion in (BEN-HAIM; TOM-TOV, 2010)). The second group of methods lack rigorous algorithmic analysis but relies on heuristics to provide empirical results for query accuracy and reduced memory usage. Examples of methods in this group are the GK sketch (GREENWALD; KHANNA, 2001), the S-Hist sketch (BEN-HAIM; TOM-TOV, 2010) moreover, the t-digest by Dunning (DUNNING; ERTL, 2014).

Data Structures for Streaming Data. Data structures need to dynamically process streams of geospatial data while enabling the fast execution of spatiotemporal queries, such as the *top-k* query that ranks and returns only the k most relevant data matching predefined spatiotemporal criteria. One approach is to store data continuously in a dense array following the order given by a space-filling curve, which leads to desirable data locality. Inserting an element takes on average $O(n)$ data movements, i.e., the number of elements to move to make room for the newly inserted element. The cost of memory allocations can be reduced using an amortized scheme that doubles the size of the array every time it gets full. However, elements are often inserted in batches in an already sorted array. In that case, one approach is to use adaptive sorting algorithms to take advantage of already sorted sequences (ESTIVILL-CASTRO; WOOD, 1992; COOK; KIM, 1980; MCGLINN, 1989). Timsort (PETERS, 2002) is an example of an adaptive sorting algorithm with efficient implementations. Another possibility is to rely on trees of linked arrays. The B-tree (BAYER; MCCREIGHT, 1972) and its variations (BRODAL; FAGERBERG, 2003) are probably the most common data structure for databases. The UB-Tree is a B-tree for multidimensional data using space-filling curves (RAMSAK et al., 2000). These structures are seldom used for in-memory storage with a high insertion rate. They are competitive when data access time is large enough compared to management overheads, often the case for on-disk storage. Such data structures are *cache-aware*, *i.e.*, to ensure cache efficiency they require a calibration according to the cache parameters of the target architecture.

Sparse arrays are an alternative that lies in between dense arrays and trees of linked arrays. Data is stored in an array larger than the actual number of elements to store, using the extra room to make insertions and deletions more efficient. Itai et al. (ITAI; KONHEIM; RODEH, 1981) were probably the first to propose such data structure. Bender et al. (BENDER; DEMAINE; FARACH-COLTON, 2005; BENDER; HU, 2007a) refined it, leading to the Packed Memory Array (PMA). The main idea is that by maintaining a controlled spread of gaps, insertions of new elements can be performed moving much fewer than $O(N)$ elements. The insertion of an element in the PMA only requires $O(\log^2(N))$ amortized element moves. This cost goes down to $O(\log(N))$ for random insertion patterns. Bender and Hu (BENDER; HU, 2007a) also proposed a more complex PMA, called adaptive PMA, that keeps this $O(\log(N))$ for specific insertion patterns like bulk insertions. PMA is a *cache-oblivious* data structure (FRIGO et al., 1999), *i.e.* it is cache efficient without explicitly knowing the cache parameters. Such data structures are interesting today since the memory hierarchy is getting deeper and more complex with different block sizes. Cache-oblivious data structures are seamlessly efficient in this context. Bender et al. (BENDER; DEMAINE; FARACH-COLTON, 2005; BENDER et al., 2007) also proposed to store a B-tree on a PMA using a van Emde Boas layout, leading to a cache-oblivious B-tree. However, it leads to a complex data structure without a known practical implementation. Still, PMA has few known applications. Mali et al. (MALI et al., 2013) used PMA for dynamics graphs. Durand et al. (DURAND; RAFFIN; FAURE, 2012) relied on PMA to search for neighbors in particle-based numerical simulations. They indexed particles in PMA based on the Morton index computed from their 3D coordinates. They proposed an efficient scheme for batch insertion of elements, while Bender relied on single element insertions.

Stream Processing. Stream processing engines, like GeoInsight for MS SQL StreamInsight (KAZEMITABAR et al., 2010), are tailored for single-pass processing of the incoming data without the need to keep in memory a large window of events that require an advanced data structure. The emergence of geospatial databases led to the development of a specialized tree, called R-Tree (GUTTMAN, 1984), that associates a bounding box to each tree node. Several data processing and management tools have been extended to store geospatial data relying on R-trees or variations like the SpatiaLite (SpatiaLite, 2019) extension for SQLite or PostGis (PostGIS, 2019) for PostgreSQL. Though such spatial libraries brought flexibility for applications in the context of traditional spatial databases, their algorithms are not adapted to consume a continuous data stream. Magdy et al. (MAGDY et al., 2014; MAGDY et al., 2016) proposed an in-memory data structure to query and update real-time streams of tweets. Initially called Mercury, then Venus and eventually Kite (MAGDY; MOKBEL, Mars 2017) for the latest implementation (Kite is also benchmarked in our experiments). They rely on a pyramid structure that decomposes the space into H levels. Periodically the pyramid is traversed to remove the oldest tweets to keep the memory footprint below a given budget. This idea to rely on bounding volume hierarchies is also popular in computer graphics for indexing 3D objects and accelerating collision detection (YOON; MANOCHA, 2006). One difficulty in these data structures is to ensure fast insertions while keeping the tree balanced. The data structure may also become too fragmented in memory leading to an increase of cache misses. The partitioning criteria are based on heuristics. There is often no theoretical performance guarantees.

1.2 Collection of Papers and Contributions

This thesis consists of the collection of papers published by the author during the period of his Ph.D. The subsections below summarize the contributions and most relevant points of each paper in the collection. Each paper comprises a following separate chapter.

1.2.1 Real-Time Visual Exploration of Big Data

This research started with the proposal of a novel concept to avoid the large memory footprint common to data cubes by introducing ***pivots*** (PAHINS; COMBA, 2016). It led to a solution of representing hierarchical and flat data structures commonly used to accelerate data queries, and that can be used to generate well-known visual encodings such as binned scatter plots, histograms, and heat maps. This work provides algorithms to support a collection of queries over aggregated data, such as counting events in a particular spatial region, categorical queries associated with selections, and temporal queries of any granularity. Memory and timing measurements using a variety of synthetic and real-world datasets are also reported. The *pivot* concept is described in Chapter 2.

The following work improves the capabilities of the *pivot*, and proposes the **Hashedcubes** (HC) (Pahins et al., 2017), a data structure that enables real-time visual exploration of large datasets that, on the date of publication, improved the state of the art by its low memory requirements, low query latencies, and implementation simplicity. In some instances, HC notably requires two orders of magnitude less space than other data cube visualization proposals. Algorithms to build and query HC, and how it can drive well-known interactive visualizations such as binned scatterplots, linked histograms, and heatmaps are presented. Hashedcubes is described in Chapter 3.

1.2.2 Similarity-based Visual Exploration of Multidimensional Datasets

Big data visualization is the main task for data analysis. Due to its complexity regarding volume and variety, very large datasets are unable to be queried for similarities among entries in traditional Database Management Systems. This work proposes **SimilarityCubes** (SC) (PERALTA et al., 2018), an effective approach for indexing millions of elements with the purpose of performing single and multiple visual similarities queries on multidimensional data associated with geographical locations. It introduces an approach that makes use of the Z-Curve algorithm to map into 1D space considering similarities between data. SimilarityCubes is described in Appendix A.

1.2.3 Real-Time Visual Exploration and Analysis Based on Order Statistics

This research proposed data structures to perform interactive visual exploration of large datasets, but, while powerful, these approaches overlooked an essential aspect of data analysis: the inherent uncertainty due to data aggregation. This work introduces **Quantile Data Structure** (QDS) (de Lara Pahins; Ferreira; Comba, 2019), a data structure that bridges this gap by supporting interactive uncertainty visualization and exploration based on order statistics. The idea behind this method is that while it is not possible to *exactly* store order statistics in a data cube, it is indeed possible to do it *approximately* To achieve this, Quantile Data Structure makes use of an efficient non-parametric distribution approximation scheme called p-digest. Both QDS and p-digest are described in Chapter 4.

1.2.4 Visual Formation and Comparison of Patient Cohorts

The increasing availability of large-scale health-care data in various sectors, medical experts, need effective methods to identify patient cohorts, examine and explain their health and its evolution, and compare cohorts. Medical cohort analysis exhibits the collective behavior of patients, providing insights on the evolution of their health conditions and their reaction to treatments and their environment. This work proposes **COVIZ**, an interactive system that lets medical experts form cohorts, obtain their various statistics, examine their health condition and treatments, visualize how their health evolves, and compare cohorts. COVIZ is described in Chapter 5.

1.2.5 Real-Time Visual Exploration of Streaming Big Data

The visual analysis of large multidimensional spatiotemporal datasets poses challenging questions regarding storage requirements and query performance. This research proposed data structures to address these problems that rely on indexes that pre-compute different aggregations from a known-a-priori dataset. Consider now the problem of handling *streaming* datasets, in which data arrive as one or more continuous data streams. This work introduces **Packed-Memory Quadtree** (PMQ) (TOSS et al., 2018), a novel data structure designed to support the visual exploration of streaming spatiotemporal datasets. Packed-Memory Quadtree is *cache-oblivious* to perform well under different cache configurations. This work is validated under different dynamic scenarios and compared to other recent strategies. PMQ is described in Appendix B.

2 HASHEDCUBES: A DATA STRUCTURE FOR REAL-TIME EXPLORATION OF LARGE MULTIDIMENSIONAL DATASETS

Published in: Proceedings of the 1st Workshop on Data Systems for Interactive Analysis, 2015.
Authors: Cícero A. L. Pahins and João L. D. Comba.

2.1 Abstract

The proliferation of statistics, application usage records, GPS and other devices, lead to large and complex volumes of data. Visualization techniques are used to perform analysis of this data, and the creation of different associations over this data allow to discover patterns that would hardly be recognized by individual data analysis. The analysis of big data presents challenges which frequently requires aggregation of the data. Traditional tools like relational databases, or statistical and visualization applications, are not appropriate to the volume of data presented, and the well-known data cube aggregation operation can take a prohibitively large amount of space. We propose HashedCubes, a data structure that has low memory requirements and enables interactive exploration and analysis of complex and large multidimensional datasets. We present algorithms to build and query HashedCubes, as well as how it can be used to generate well-known visual encodings such as binned scatter plots, histograms, and heat maps. We shown memory and timing measurements using a variety of synthetic and real-world datasets ranging from 4.7M to 1B records.

2.2 Introduction

Data acquisition has never been so broad, diverse, and accessible. Nowadays, almost every computer-based gadget offers a broad range of options to collect multidimensional data. Social networking, government data, application records and many others generate immense amounts of raw data. The exploration and analysis through aggregation of these datasets is a valuable opportunity to researchers, that often find tools, such as histograms and heat maps, best suited for the task.

Traditional tools such as relational databases, or statistical packages, have difficulty to handle large multidimensional datasets. Naive solutions can take prohibitively large amounts of memory or time to answer as the number of dimensions increases. Thus, specialized data structures to accelerate query time in these datasets are necessary.

Data aggregation enables the design of sophisticated visualization tools once queries are performed on preprocessed data. In this way, a large number of recent solutions extend the well-known data cube (GRAY et al., 1997b), a data structure that aggregates all data dimensions in a regular n-dimensional structure. Its main disadvantage is that memory usage becomes impractical when the number of dimensions increases.

The *imMens* approach described in (LIU; JIANG; HEER, 2013) combines data reduction, multivariate data tiles, and parallel query processing, aiming to minimize the *data cube* memory usage. Its multivariate data tile methods are based on the observation that for any pair of 1D or 2D binned plots, the maximum number of dimensions needed to support brushing and linking is four. Thus, an n-dimensional data cube can be decomposed into a collection of smaller 3- or 4-dimensional projections. Furthermore, these decomposed data cubes are segmented into multivariate tiles, like the ones used by Google Maps, but the approach lacks support for compound brushing of more than four dimensions.

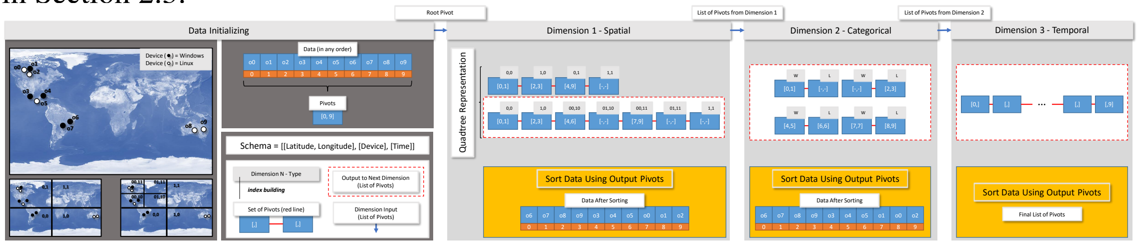
Nanocubes (Lins; Klosowski; Scheidegger, 2013) is a compact variation of a *data cube* that can handle a large number of dimensions. It defines a search key that is used to combine aggregations of independent dimensions at varying levels of detail, but support only spatiotemporal datasets. *HashedCubes* is an alternative to *Nanocubes* that allows a more compact and expressive representation through the extensive use of pivots. The notion of pivots described in (MORA, 2011; PAHINS; POZZER, 2014) allow fast queries through compact and linear storage.

In this work, we introduce *HashedCubes*, a novel data structure to accelerate queries from interactive visualization tools that explore and analyze large multidimensional, spatiotemporal, datasets. HashedCubes supports spatial queries, such as counting events in a particular spatial region, categorical queries associated with selections, and temporal queries of any granularity. On average, queries are performed under 30 milliseconds for a single thread execution. In particular, we show how to build HashedCubes to fit in the main memory of a personal computer configuration.

HashedCubes offers a supporting infrastructure to real-time interactive visualizations systems through binned aggregation. In summary, our main contributions are:

- a novel data structure that enables real-time exploratory visualization of large mul-

Figure 2.1: HashedCube building steps for ten points [o0, ..., o9]. The process is described in Section 2.3.


Root Pivot
List of Pivots from Dimension 1
List of Pivots from Dimension 2
Data Initializing
Dimension 1 - Spatial
Dimension 2 - Categorical
Dimension 3 - Temporal
Data (in any order)
Pivots
Schema = [[Latitude, Longitude], [Device], [Time]]
Dimension N - Type
index building
Output to Next Dimension (List of Pivots)
Set of Pivots (red line)
Dimension Input (List of Pivots)
Quadtree Representation
Sort Data Using Output Pivots
Data After Sorting
Final List of Pivots

tidimensional, spatiotemporal datasets;

- a pivot schema to represent hierarchical and flat data structures commonly used to accelerate data queries;
- experiments to measure memory usage and building time of our method on synthetic and real-world datasets.

2.3 HashedCubes Concept

Hierarchical data structures can store information in a myriad of ways. One possibility is to store information only at the leaves of the tree. However, to recover information at an internal node it is necessary to aggregate information in a bottom-up fashion from the leaves up to the given node. In the case of applications that require queries at varying levels of the tree, this approach becomes expensive. The replication of information at each node may become too expensive, especially if the information is complex. Finding a balance in these alternatives is the goal of HashedCubes.

Fractional Cascading (CHAZELLE; GUIBAS, 1986) is an approach that explores the property that information stored at a given node of the tree is a subset of the information stored at ancestral nodes. By using this property, range queries in higher dimensions are made more efficient. Variations of this idea appear in several works. One interesting approach is to keep a single array containing all the information ordered by a depth-first traversal of the tree. Such ordered array has the property that the information associated with each node of the tree is stored in a contiguous fashion. Therefore, it suffices to keep a pair of markers in this array to represent the information stored at each node. This pair of markers is called a *pivot*, and has been used in (MORA, 2011; PAHINS; POZZER, 2014).

A pivot represents an interval $[i_0, i_1]$, where i_0 and i_1 are the initial and final markers within a given array. In HashedCubes, a pivot is used to delimit a list of elements with a common semantic value, such as the same spatial quadrant, categorical attribute or temporal bin. The difference between the initial and final values enables the trivial computation of the set size. We use pivots to delimit sets in a hierarchical fashion, in such way that the size of aggregations can be pre-computed.

Given a multidimensional dataset, a linear array called *Hash* is associated with a root pivot $[0, n-1]$ and indicates the universe of n elements. Each element of the *Hash* array is an integer that points to a position in the dataset. Initially, *Hash* array is stored in any order, as shown in Figure 2.1.

HashedCubes supports three distinct dimension types: spatial, categorical and temporal, which can be build and traversed in any order. Since lower nodes in the hierarchy are subsets of higher nodes in the hierarchy, the list of pivots in a given dimension also represent subsets for that given dimension.

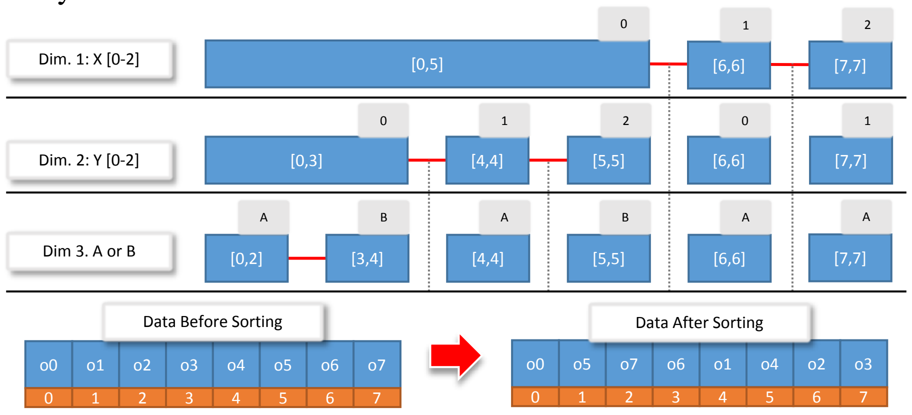
Let's assume that we are building a categorical dimension that can have values A or B. The construction algorithm processes all values delimited by a given pivot. For the root of the tree, the categorical dimension output will be a list of pivots with two elements that represent the separation of the categories A and B. As the pivot represents the initial and final positions of a set of values, the entries (in fact the *Hash* array) must be ordered to reflect this organization. This can be seen in Figure 2.1 - *Sort Data Using Output Pivots*.

Any of the dimensions can receive a list of pivots as input. Every pivot in this list is further refined as necessary to create lists of subset pivots. A collection of all generated lists of pivots is the output obtained after processing each dimension.

Pivot hierarchy is a core concept of HashedCubes. Consider the HashedCubes in Figure 2.2. The illustration shows a dataset that contain eight records, distributed along the X and Y axis, from values 0 to 2. Each record has a categorical attribute A or B associated with the spatial information. Observe the pivot $[0, 5]$ from dimension 1. After the data array is sorted, all elements that are distributed along $X = 0$ are located between positions 0 and 5. Moreover, dimensions 2 and 3 have pivots that are contained in this range, and, therefore, represent subsets with the same X value. In contrast, the pivot $[6, 6]$ is repeated three times across the HashedCube. It indicates a set that all elements have the same bins for each of the dimensions.

The pivot hierarchy mimics the tree hierarchy since each pivot represents a set that can be further divided into a variable number of subset pivots. Therefore, it allows the query algorithm to skip dimensions with no selection or constraint. Nanocubes (Lins;

Figure 2.2: An illustration of how to interpret pivot hierarchy. HashedCubes reassembles tree hierarchy.



Klosowski; Scheidegger, 2013) has a similar approach called shared pointers, but incurs in additional memory cost. Furthermore, the difference between initial and final values of a pivot represents the set size. Thus, pivot hierarchy enables the interpretation of this property as precomputed aggregation sizes in any refinement.

Note that HashedCubes reassembles tree hierarchy, but differently from these data structures, it does not store edges between one dimension to another. Siblings pivots (nodes) are stored as lists (red lines in Figure 2.1 and 2.2). Each dimension stores collections of pivots.

2.4 HashedCubes Construction

To build HashedCubes, firstly it is necessary to define an indexing scheme that encodes the ordering that dimensions are processed from the dataset (e.g. first spatial, then categorical, and finally temporal). For every dimension of the indexing scheme, the algorithm tags the respective bin of each object. Then, a sorting algorithm is executed in the predefined spaces delimited by a list of broader pivots. Initially, this list contains only the root pivot, which indicates the dataset universe space. After that, narrow pivots are generated and represent subsets. The result of processing each dimension, a list of pivots, is the input to the next dimension (that are subsets of the above).

Note that, in contrast to data cubes (GRAY et al., 1997b) or Nanocubes (Lins; Klosowski; Scheidegger, 2013), HashedCubes does not perform aggregations across every possible set of dimensions. Instead, our method leverages the pivot hierarchy to compute the faulty aggregations on-the-fly. Take as example the illustration on Figure 2.2. Root pivot $[0, 7]$ aggregates every object of the dataset universe. However, how to pro-

Table 2.1: Overall summary of the relevant information for building HashedCubes. *Not Measured.

dataset	objects	memory	time	pivots	schema
brightkite	4.7 M	26 MB	4 s	1.9 M	lat & long, hour of day (24), day of week (7), time
gowalla	6.4 M	20 MB	5 s	1.5 M	lat & long, hour of day (24), day of week (7), time
flights	121 M	208 MB	128 s	17.3 M	lat & long, departure delay (9), carrier (29), time
splom-10	1 B	120 KB	NM*	15 K	d1 (10), d2 (10), d3 (10), d4 (10), d5 (10)
splom-50	1 B	66 MB	NM*	8.2 M	d1 (50), d2 (50), d3 (50), d4 (50), d5 (50)

ceed if we want to know how many objects are labeled as A or B in the categorical dimension 3? Taking advantage of the linear storage used by HashedCubes, we can trivially combine aggregates from multiple branches. Thus, with only one pass, we compute aggregations of any dimension. This algorithm is described in Section 2.5 and it is responsible for significant memory savings when compared against existent solutions, enabling the exploration of very large datasets.

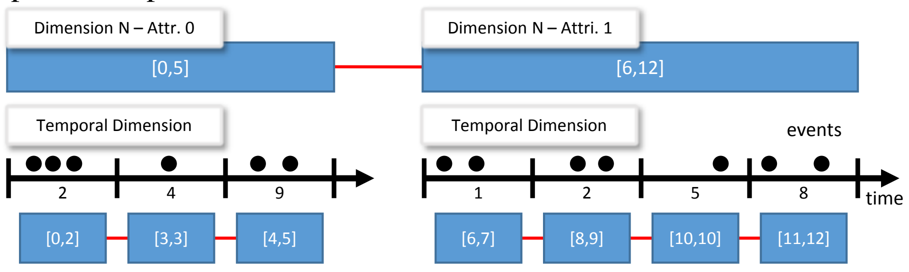
2.4.1 Spatial Dimensions

Spatial attributes usually require space partitioning data structures, which are often hierarchical. HashedCubes represents hierarchical data structures by storing delimiting lists of pivots for each refinement step. As a result, the output is the aggregation of all leaf nodes or end values of the represented structure. Spatial queries broken down by latitude and longitude are the basis for heat maps and choropleth maps.

It is helpful to think of spatial dimensions as being represent by quadtrees. Each node is associated with a pivot that delimits the objects within that space. If a query matches the exact region represented by a node, then the pivot represent all aggregates that refers to that set. Otherwise, we compute the minimal disjoint set of nodes that cover the query region. The region visible on-screen can be interpreted as spatial queries, reducing the total processing necessary. Our current implementation uses Mercator projection for consistency with existing map tile providers, such as OpenStreetMap (HAKLAY; WEBER, 2008).

Typically, map tiles providers use coordinates (x, y, z). The tuple $[x, y]$ are integer addresses, and z represents the zoom level, that in most cases range between 0 (zoomed out) and 18 (zoomed in). Each zoom increment doubles $[x, y]$ resolution, and consists of 2^{2n} or 4^n tiles. In this manner, our quadtree implementation is limited to a maximum of 26 levels of divisions, the maximum zoom value plus 8, which denotes how many levels to break down tile space.

Figure 2.3: Spatial dimension indexing scheme. A period of time is represented by a timestamped list of pivots.



2.4.2 Categorical Dimensions

Categorical attributes of multidimensional datasets are usually divided into specific values or ranges. Thus, the resulting index should be lists of pivots that represents data grouping in regions defined for each of these bins. Categorical queries are the basis for histograms, binned scatter plots and parallel sets (KOSARA; BENDIX; HAUSER, 2006), each of these requiring a different breakdown from HashedCube search engine results.

To build the index, each categorical dimension has a specialized *Comparator Function*. A comparator function computes, for each dataset element, a position in the resulting list of pivots. More specifically, it compares an element against all dimension attributes and returns a bin tag. Let's take as example the illustration of Figure 2.2. Dimension 3 has a comparator function that returns, for any dataset element, either A or B. Thus, the output is ordered based on the tags and to reflect the pivot hierarchy.

2.4.3 Temporal Dimensions

To represent temporal dimensions, we take advantage from pivots property of representing set sizes. Time series are stored as timestamped lists of pivots. Each pivot represents a bin. To ease the exposition, let's take as example a HashedCube in which time series are binned per day since epoch time. The building algorithm will compute the bin of every dataset element. This collection of bins will represent a sparse list of sets. From that, will be generated a dense list of timestamped bins, as illustrated in Figure 2.3.

This schema enables to store time series from any granularity without requiring a nested data structure. Moreover, since each pivot is associated with a timestamp, it allows to compute the number of events along any contiguous period by finding the pivot with the least upper bound and greatest lower bound from the period of time. All three dimension

Table 2.2: Subset of queries supported by HashedCubes.

Query	URL
count of dimension	**/field/**<category>
count along period	**/tseries/**<from>/<to>/<incr>
subset of dimension	**/where/**<category>=<value$_0$> |...|<value$_n$>
heatmap of region	**/region/**<level>/<lon>/<lat>/<widht>/<height>

types has input and output as lists of pivots.

2.5 HashedCubes Queries

Initially, the query range is the dataset universe represented by the pivot $[0, N]$. The query result in each dimension is a delimiting list of pivots of the selected data, thus, this lists becomes the new range query. This process is interactively repeated until the last dimension. Note that, unlike other data structures for real-time exploration and analysis of large datasets, such as Nanocubes (Lins; Klosowski; Scheidegger, 2013), our method is not tree-based, so data iteration occurs in linear memory. Such approach offers an appealing performance, since the CPU cache automatically optimizes burst memory operations (GOODMAN, 1983; KRISHNAMOHAN; FARMWALD; WARE, 1996).

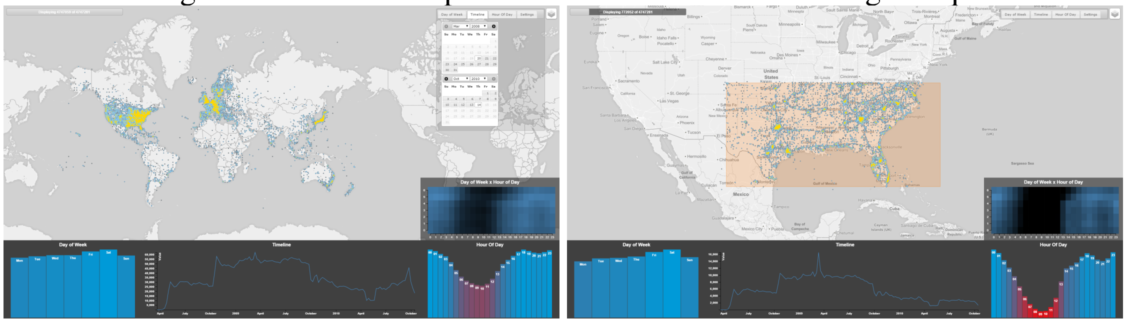
Our method optimizes query time by the clever use of pivots. Take as example the schema in Figure 2.2. As already known, inferior dimensions represent subsets of superior dimensions, thus, if a query value does not select any attribute from a particular dimension, such as Dimension 1 (Figure 2.2), the query algorithm will simply skip it until finding a valid selection. The skipping dimension process provides a significant performance gain when queries select all or none attributes from dimensions. Selecting all attributes has the same effect that selecting none.

Our method supports three types of query, which have different complexity in traversing the data structure: Querying a Value, Querying a Value with a List of Pivots (range) and Querying a List of Values with a List of Pivots (range). Each query function must be picked according to the type of expected pivots selection. The correct choice of the functions ensures minimal query time.

2.6 Experiments and Discussion

We use a client-server architecture for the current implementation of our method. The server reads multidimensional data, builds HashedCubes and then waits for queries

Figure 2.4: HashedCubes enables real-time exploratory interactive systems using a wide range of visual encodings, such as heat maps and choropleth maps, histograms, binned scatter plots, parallel sets, and others. They support brushing & linking across any dimension. The images show the heatmap and charts associated with two given queries.



from the client. The server is implemented under Java EE technology, using Glassfish as back-end. It is easy to plug in different data structures for each dimension since we use Java generics to allow our method to operate on objects of various types. Memory usage and query response time were optimized.

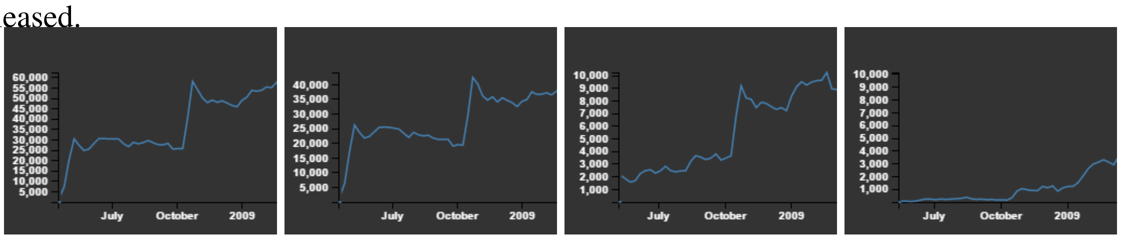
Our server exposes its API via HTTP using Jersey, an open source implementation of JAX-RS Java API, as shown in Table 2.2. Queries can be combined to generate a wide range of visual encodings, such as heat maps and choropleth maps [region]+[where], histograms [field]+[region]+[where], scatter plots [field]+[field]+[region]+[where], and others.

The server is easily parallelizable since the data structure are no longer mutated after building. In the front-end, we develop a browser-based client written in Javascript, SVG, and HTML5. Server queries are asynchronous through the use of jQuery Ajax API. Leaflet, Heatmap.js, jQuery UI, D3.js and other javascript libraries were combined to generate the real-time exploration and visualization features, as shown in Figure 2.4.

For the experiments, we paid particular attention to how much memory was required to store HashedCubes, which varied considerably from one dataset to another, as shown in Table 2.1. Our dataset selection enables a direct comparison against other state-of-the-art solutions. Brightkite (4.7M) and Gowalla (6.4M) are two location-based social networks that let users share their locations. Both were used by $imMens$ and $Nanocubes$. Flights dataset (121M) tracks the on-time performance of domestic flights by U.S. air carriers and was used to by $Nanocubes$. SPLOM (1B) is a synthetic dataset that was designed to stress data cube technology. It was used by $imMens$ and $Nanocubes$.

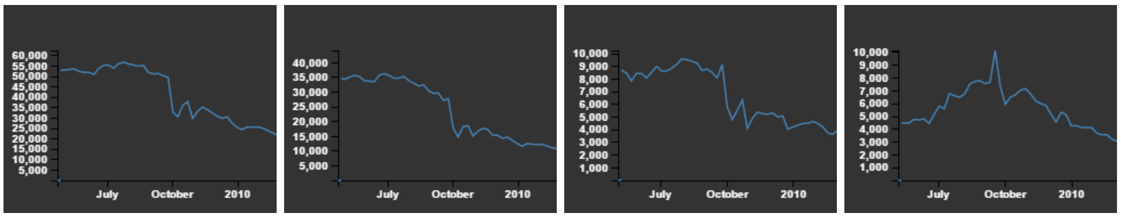
Since the building time was a relevant factor of HashedCubes, the construction algorithm has been optimized for speed by avoiding repeated memory allocations and deallocations (using techniques for avoiding automatic Java GC collections). We find that building time is dominated by sorting time steps. To measure query time, we performed

Figure 2.5: HashedCubes temporal and spatial indexing schemes allow the exploration and analysis of check-ins as global trends, as well as geographically restrict events. In October 2008 Brightkite Iphone app goes live. In October 2009 Brightkite 2.0 was released.



Global United States of America Western Europe Japan

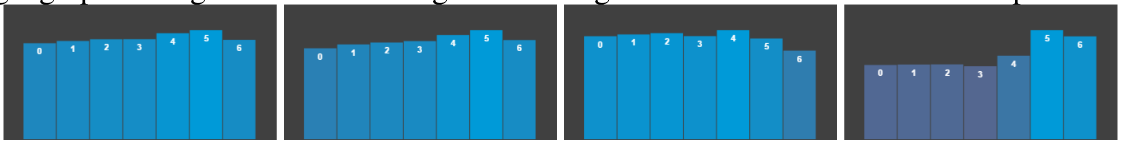
(a) October 2008



Global United States of America Western Europe Japan

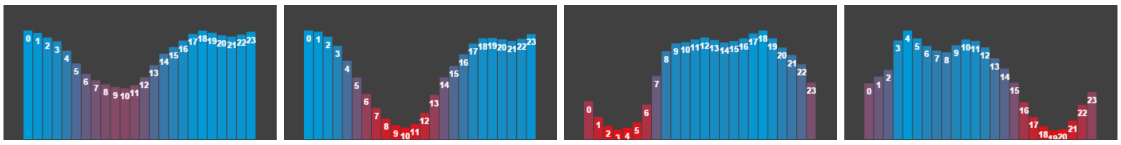
(b) October 2009

Figure 2.6: By indexing Day of Week and Hour of Day as categorical dimensions of the Brightkite dataset, HashedCubes enables to highlight user interactions across specific geographical regions. There is a significant usage difference between US and Japan.



Global United States of America Western Europe Japan

(a) Day of Week



Global United States of America Western Europe Japan

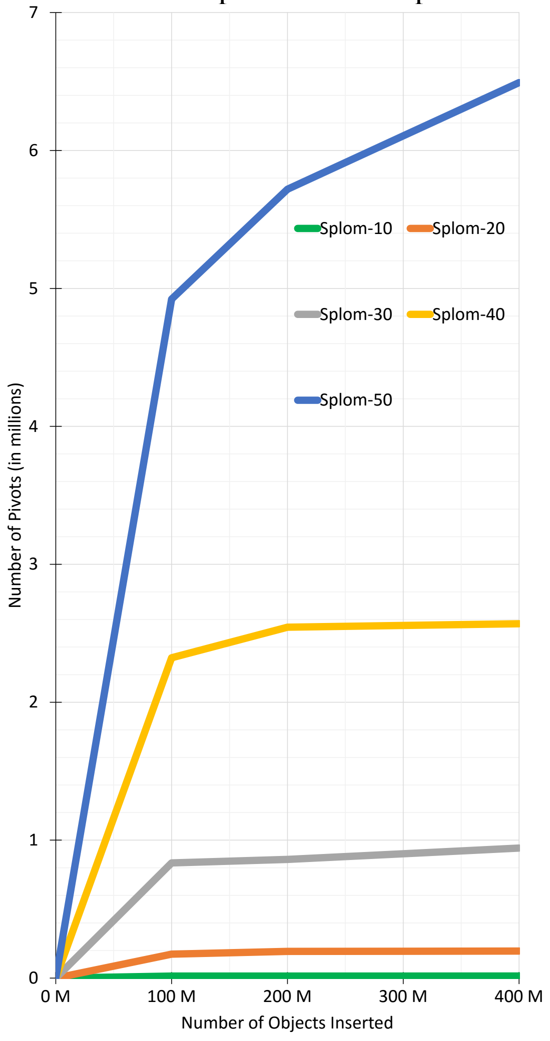
(b) Hour of Day

brushing & linking across dimensions. All three real-world datasets consistently presented query times under $30ms$ for various rollups and drill down test combinations. The network bandwidth between server and client interfaces were dominated by transference of geographical tiles information.

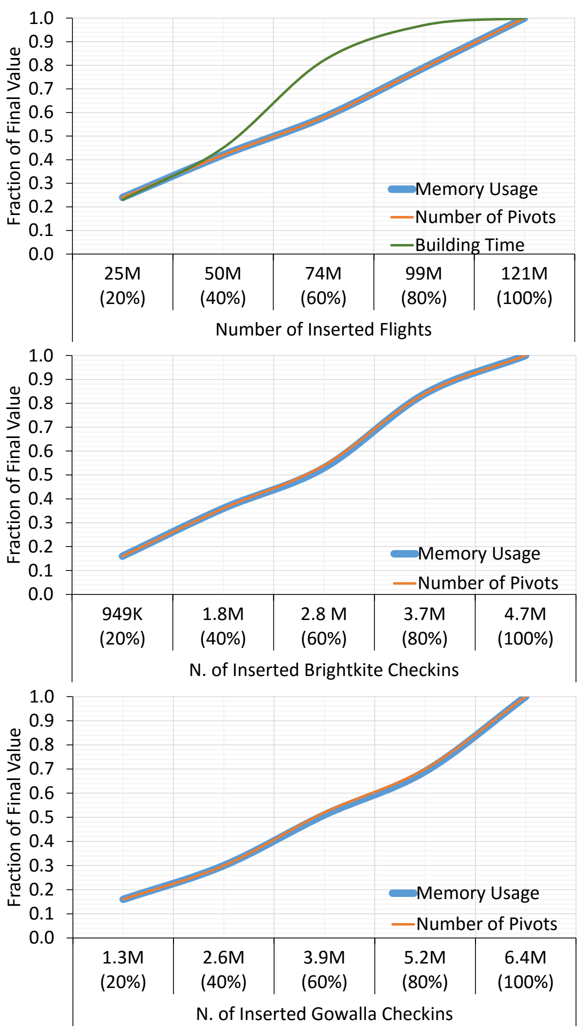
2.6.1 Result Analysis

HashedCubes indexing scheme enables to interactively explore large multidimensional datasets by supporting a collection of queries over the aggregated data, such as

Figure 2.7: (a) Growth of the number of pivots when inserting objects into HashedCubes. Notice the *key saturation* effect. (b) HashedCubes memory usage is directly proportional to the number of pivots. From top to bottom: Flights, Brightkite and Gowalla datasets.



(a)



(b)

counting events in a particular spatial region, categorical queries associated with selections, and temporal queries of any granularity. As showed in Section 2.6, queries can be combined to generate a wide range of visual encodings. Based on that, researchers are more willing to spend time exploring and analyzing complex data sets, what can lead to discovery of hidden and interesting patterns, as shown in Figures 2.5 and 2.6.

From the performance perspective, Figure 2.7b shows curves for memory usage and number of pivots for Flights, Brightkite, and Gowalla datasets using HashedCubes. As additional information, Flights graph shows building time. All metrics are relative to the final HashedCube numbers presented in Table 2.1. For each dataset, inserted records ranged from 20% to 100% of the total.

All real datasets have records over wide periods of time. As a result, the growth in the number of pivots is proportional to the growth of the temporal dimension. Records are sorted by timestamps, which translates to a constant insertion rate of distinct temporal values. In our experiments, we used the time resolution set to one hour, and, therefore, two records with timestamps 15h05m and 15h55m have pivots with the same time label. Most records will not require more memory since their pivots were already inserted into the HashedCube index.

Memory usage is directly proportional to the number of pivots, i.e., to the number of used bins per dimension. This is due to the fact that HashedCubes do not spend memory to store edges (pointers) between nodes from one dimension to another, since each is independent.

When HashedCubes memory usage is directly compared against the state-of-the-art Nanocubes (Lins; Klosowski; Scheidegger, 2013), we find a reduction factor of up to 61x in the best case. Building the HashedCubes for brightkite and flights real-world datasets, requires 26MB and 208MB of memory, respectively. For the same dimension schema, Nanocubes requires 1.6GB and 2.3GB. Keep in mind that HashedCubes does not perform aggregations across every possible set of dimensions, it computes faulty aggregations on-the-fly. Tree hierarchy is reassembled by pivots hierarchy interpretation, actually enabling the removal of any dimension from the index after building.

Figure 2.7a shows pivots growth for the SPLOM dataset ranging from zero to four hundred million inserted records into HashedCubes of different bin size. Every dimension of this dataset is collect from synthetic generators that have normal distribution, which means that the set of high probability values are quickly sampled, making harder for new records with an unseen bin. It highlights an effect known as *key saturation*. Due to the key saturation effect, most inserted records does not require additional memory since their pivots were already present in the HashedCube index, a phenomenon that performs an important role to reduce memory usage.

2.7 Conclusions and Future Work

In this paper we presented HashedCubes, a fast, easy to implement and, memory efficient data structure to answer queries from interactive visualization tools that explore and analyzes large multidimensional datasets. The pivot approach enables traversal in any order and allows to include multiple spatial dimensions into the index, useful so that one could visualize, for example, datasets with two natural geographical locations, such as phone calls.

Our major contributions have shown that (i) is possible to represent hierarchical and flat data structures using an optimized pivot schema that is stored in a linear fashion way, and (ii) demonstrated that this leads to memory savings over other solutions, as shown in Section 2.6.

Taking advantage of the performance given by HashedCubes, researchers can develop richer, more appealing and seamless interactive visualizations tools. As future work, extension of pivots concept, as well as further query optimizations, should be performed.

3 HASHEDCUBES: SIMPLE, LOW MEMORY, REAL-TIME VISUAL EXPLORATION OF BIG DATA

Published in: IEEE Transactions on Visualization and Computer Graphics (TVCG), 2017.
DOI: 10.1109/TVCG.2016.2598624
Qualis CAPES: A1
Authors: Cícero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger and João L. D. Comba.

3.1 Abstract

We propose Hashedcubes, a data structure that enables real-time visual exploration of large datasets that improves the state of the art by virtue of its low memory requirements, low query latencies, and implementation simplicity. In some instances, Hashedcubes notably requires two orders of magnitude less space than recent data cube visualization proposals. In this paper, we describe the algorithms to build and query Hashedcubes, and how it can drive well-known interactive visualizations such as binned scatterplots, linked histograms and heatmaps. We report memory usage, build time and query latencies for a variety of synthetic and real-world datasets, and find that although sometimes Hashedcubes offers slightly slower querying times to the state of the art, the typical query is answered fast enough to easily sustain a interaction. In datasets with hundreds of millions of elements, only about 2% of the queries take longer than 40ms. Finally, we discuss the limitations of data structure, potential spacetime tradeoffs, and future research directions.

3.2 Introduction

Designers of interactive visualization systems face serious challenges in the presence of large, multidimensional datasets. On one side, naive implementations of repeated linear scans of the dataset of interest no longer offer acceptable latencies: this makes simple data structures no longer attractive. On the other side, sophisticated implementations of precomputed indices built specifically for visualization have been proposed recently.

These offer attractive query times, but their implementations are not trivial to integrate with existing systems, require GPU support, or have another similar downside. This paper provides an affirmative answer to the following question: is there a simple data structure that offers much of the performance of the more sophisticated indices, while maintaining a relatively-low memory footprint and implementation simplicity?

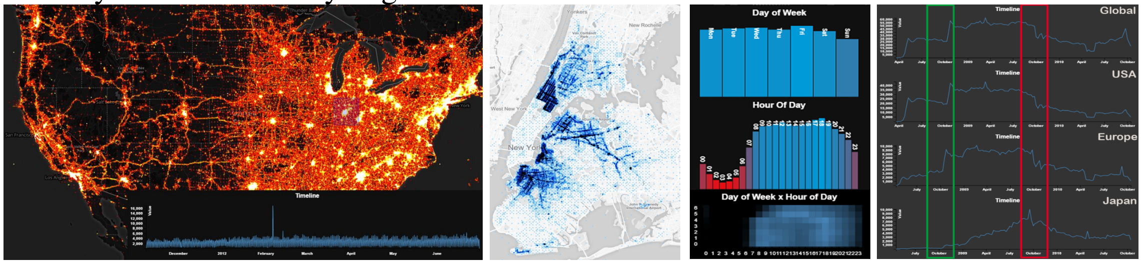
Specifically, we present *Hashedcubes*, a novel data structure that enables fast querying for interactive visualizations of large, multidimensional, spatiotemporal datasets. Hashedcubes supports spatial queries, such as counting events in a particular spatial region; categorical queries over subsets of attribute values; and temporal queries over intervals of any granularity. As we report on Section 3.7, a typical query is returned in under 30 milliseconds in single-threaded execution. As a practical matter, Hashedcubes was designed to target the amount of main memory of a modern desktop or laptop personal computer (on the order of 16 to 32GB of main memory). In summary, this paper contributes:

- a simple data structure for real-time exploratory visualization of large multidimensional, spatiotemporal datasets, advancing the state of the art especially with respect to implementation simplicity and memory usage,
- an experimental validation of a prototype implementation of Hashedcubes, including a suite of experiments to assess query time, memory usage, and build time of the data structure on synthetic and real-world datasets, and
- an extended discussion of the trade-offs enabled by Hashedcubes, including limitations and open research questions.

3.3 Related Work

In this section we will focus on work directly related to interactive visual analysis of big data. For a more comprehensive list of papers, we refer the reader to the surveys on big data analysis (GODFREY; GRYZ; LASEK, 2015), big data visualization (AGRAWAL et al., 2015), geospatial big data analysis (LI et al., 2015) and challenges in big data implementation (GUPTA; SIDDIQUI, 2014; MORTON et al., 2014; IDREOS; PAPAEMMANOUIL; CHAUDHURI, 2015).

Figure 3.1: Hashedcubes accelerates queries used in a wide range of interactive exploratory visualizations, such as heatmaps, time series plots, histograms and binned scatterplots, and supports brushing and linking across spatial, categorical and temporal dimensions. In this figure, we show some example visualizations backed by Hashedcubes. The left image shows 210.6 million tweets from November 2011 to June 2012, highlighting the activity during Superbowl XLVI. The central image shows 24.5 million pick-up locations of NYC green taxis rides from January 2014 to June 2015. On the right, the visualizations show different aspects of 4.5 million Brightkite check-ins, a social network. Hashedcubes balances low memory usage, fast running times, and simple implementation; it allows interactive exploration of datasets that previously either required a prohibitive amount of memory or uncomfortably large latencies.



Overview of USA tweets between Nov 2011 and Jun 2012 NYC Green Taxis pick-up Brightkite in Europe Brightkite temporal series

The need for low latency in large databases is a popular theme in the literature (ASSENT et al., 2008; SHIEH; KEOGH, 2008; CAMERRA et al., 2010). *BlinkDB* (AGARWAL et al., 2013b) builds a carefully-constructed stratified sample of the dataset, which allows interactive latencies in approximate queries over multiple terabytes of data. In essence, *BlinkDB* provides infrastructure such that Hellerstein et al.'s online aggregation has fast convergence properties (HELLERSTEIN; HAAS; WANG, 1997). *ScalarR* improves performance by manipulating physical query plans and computing a dynamic reduction of query sets based on screen resolution (BATTLE; STONEBRAKER; CHANG, 2013); it is an early, central example of explicitly taking peculiarities of a visualization setup in a DB account. *3W* is a search framework for geo-temporal stamped documents that allows fast searches over spatial and text dimensions (NEPOMNYACHIY et al., 2014). *Forecache* (BATTLE; CHANG; STON, 2015) improves performance by predicting user actions ahead of the actual queries being issued.

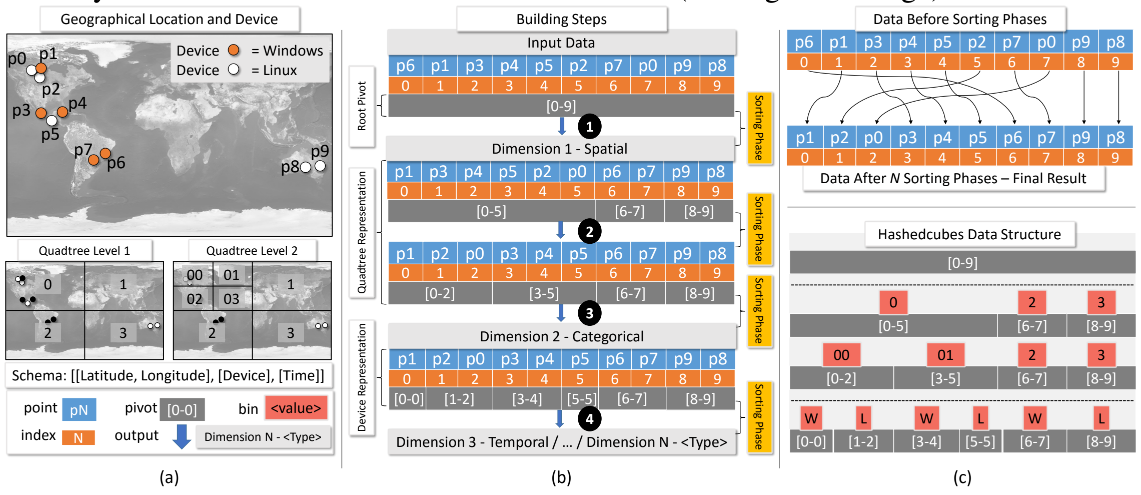
The seminal paper of Gray et al. (GRAY et al., 1997b) introduced the *data cube* concept, which laid the foundation for many other methods (LIU; JIANG; HEER, 2013; Lins; Klosowski; Scheidegger, 2013; CAO et al., 2015), including our proposal. A data cube can be seen as a hierarchical aggregation of all data dimensions in an n-dimensional lattice. Its main disadvantage is its memory consumption, which becomes impractical as the number of dimensions increases. To address this problem, some approaches describe ways to compress data cubes, such as Dwarf (SISMANIS et al., 2002), or build on distributed databases to cope with scale requirements (KAMAT et al., 2014).

VisReduce (IM; VILLEGAS; MCGUFFIN, 2013) is an approach to data aggregation which computes visualization results in a distributed fashion. It uses a modified *MapReduce* (DEAN; GHEMAWAT, 2004) algorithm and data compression. Its main drawback is that interaction operations require on-demand aggregations. Thus, the final result is obtained only after the costly transfer over the network of partial and final aggregations. As a rule of thumb, on-demand computation is problematic for visual analysis because of latency. As Liu and Heer describe (LIU; HEER, 2014a), latencies of as little as half a second can affect the overall quality of an analyst's data exploration process. A popular alternative to hide latency is to use sampling, and report uncertainty estimates as soon as they are available (FISHER et al., 2012b). Similarly, Stolper et al. describe a general framework for progressive approach for visual analytics (STOLPER; PERER; GOTZ, 2014).

The most recent trend in research at the intersection of data management and visualizations is the explicit acknowledgement of the human perceptual system. Wu et al. suggest that database engines should explicitly optimize for perceptual constraints, by for example including the visual specification into the physical query planning process (WU; BATTLE; MADDEN, 2014). Jugel et al. offer a technique that is one such example: the query algorithms described there return approximate results which nevertheless rasterize to the same image as the exact query result would (JUGEL et al., 2014; JUGEL et al., 2016); ScalarR (BATTLE; STONEBRAKER; CHANG, 2013) is another example, mentioned earlier in this section.

Closest to Hashedcubes are *imMens* (LIU; JIANG; HEER, 2013) and *Nanocubes* (Lins; Klosowski; Scheidegger, 2013). The imMens approach combines data reduction, multivariate data tiles, and parallel query processing(using a GPU) to minimize both data cube memory usage and query latency. Its multivariate data tile methods are based on the observation that for any pair of 1D or 2D binned plots, the maximum number of dimensions needed to support brushing and linking is four. Thus, an n-dimensional data cube can be decomposed into a collection of smaller 3- or 4-dimensional projections. Furthermore, these decomposed data cubes are segmented into multivariate tiles, like the ones used by Google Maps. On the other hand, imMens lacks support for compound brushing in more than four dimensions. In comparison, Hashedcubes support any number of dimensions, even if at a potential cost in query latency. Nanocubes is a compact variation of a data cube that can handle a large number of dimensions. It defines a search key that is used to combine aggregations of independent dimensions at varying levels of detail and to max-

Figure 3.2: Overall summary for building Hashedcubes. **(a)** Input dataset of points [p_0,...,p_9] under a spatial-categorical-temporal schema. The complete process is described in Section 3.4. **(b)** Step-by-step illustration of the process for building arrays of sorted partitions, as explained in Section 3.4.2. **(c)** Data is loaded (in any order) into a sequential memory and each record is associated with an index (rectangle in orange).


Geographical Location and Device
Device = Windows
Device = Linux
Quadtree Level 1
Quadtree Level 2
Schema: [[Latitude, Longitude], [Device], [Time]]
point pN
pivot [0-0]
bin <value>
index N
output
Dimension N - <Type>
(a)
Building Steps
Input Data
Root Pivot
Quadtree Representation
Device Representation
Sorting Phase
Dimension 1 - Spatial
Dimension 2 - Categorical
Dimension 3 - Temporal / ... / Dimension N - <Type>
(b)
Data Before Sorting Phases
Data After N Sorting Phases – Final Result
Hashedcubes Data Structure
(c)

imize shared links across the data structure. *Hashedcubes* is an alternative to *Nanocubes* that eschews a large number of aggregations, allowing both a more compact representation and a much simpler implementation. Hashedcubes uses a partial ordering scheme combined with the notion of pivots (MORA, 2011; PAHINS; POZZER, 2014) to allow fast queries and a simple data structure layout.

BigVis (WICKHAM, 2013) is an R package for the visualization of large datasets and statistical modeling that can store more sophisticated event statistics of events in its bins. Hashedcubes can be extended to include the additional functionality of BigVis. The support for the visualization of origin-destination (OD) data is requested in several applications that handle trajectory data. OD Taxi data visualization (JIANG et al., 2015) and taxi trajectory data visualizations are discussed in (HUANG et al., 2016). One particularly favorable use case for Hashedcubes is in fact the visual analysis of origin-destination data. The interleaved scheme used in Hashedcubes allows sufficiently-fast queries, while requiring significantly less memory than Nanocubes and imMens.

3.4 Hashedcubes

In this section we will describe the algorithms for building and querying a Hashedcubes. Before giving the full algorithms, however, we will give some intuition on how it works. Hashedcubes combines a few different ideas, and it is easier to see how they work together by progressively building on the properties it exploits. These include hierarchical array partitions, stable sorting, and commutativity of the summaries of a list under permutations of the list.

3.4.1 Some Intuition

First, we note that the fundamental unit we want to visualize in large-scale visualizations such as heatmaps and histograms is a *count*: "how many events happened within this region at some point in time?" "How many events happened on a Tuesday?", and so on. We describe below the intuition behind answering such queries from data stored in arrays.

The following observation is trivial but important: the size of an array does not change when we shuffle it, and so we have much freedom in choosing the order of its elements. The second observation is that when data is stored in a contiguous array, there is a convenient representation for some subsets of this array: we can represent a subset S of elements from an array A by a pair of indices (b, e) such that all elements $A[i]$ for which $b \leq i < e$ are considered to belong to S. We call this pair a *pivot*. If we partition the elements of an array in a certain set of non-overlapping subsets, we can always rearrange the elements such that the chosen subsets of the partition can be represented by pivots (i.e. the subsets are contiguous along the array). In other words, we can represent a partition by permuting the array and storing the corresponding array of pivots. This representation of a partition allows us to, among other things, quickly skip large runs of the data array, while remaining simple and compact. Thirdly, this rearrangement of a partition *also* has significant freedom in its choice: as long as the partition is respected, we can choose the internal order of each subset arbitrarily. Crucially, we can think of each subset of the partition as an array in itself —after all, its elements are all contiguously stored as well— and so we can impose *further* partitions on these subsets, hierarchically. This reordering does not invalidate the first pivot representation, as long as our sorting is *stable* with respect to the first partition.

Now imagine a hypothetical network logging dataset in which we log packets that reach a particular server, and that we are interested in three attributes: day of week (d), hour of day (h), and network port (p) requested. In order to build a Hashedcubes data structure, we need to decide on an *ordering* of these attributes with which to sort the array hierarchically (note that we discuss performance consequences of these choices in Section 3.8). For this example, assume we will sort in the order we just gave. As we partition the array along each of the attributes, we store the array of pivots that represents the partitions. Note that in dimensions other than the first, this means that the finer partitions will respect the previous sorting: for example, even though all events on a Monday (or any given day of week) will be laid out contiguously in an array, not all events with a given *hour of day* will be: only the events with a given hour *and* day of week. Thus, as we go down the list of dimensions in which we are partitioning, the array of pivots becomes larger, and the partitions themselves become smaller. When the sorting process is finally finished, we will have as many arrays of pivots as there are dimensions in which we are interested in querying the dataset. In our specific case, we will have three pivot arrays: one for the d partition, one for the (d, h) partition, and one for the (d, h, p) partition.

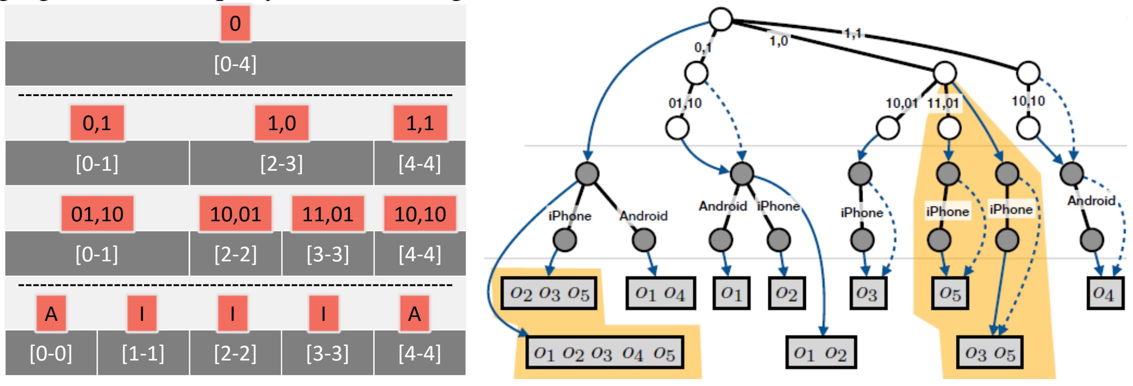
How does this hierarchical sorting help answer queries quickly? For example, if we are interested in plotting a histogram of requests in which bins represent different hours of the day, it is clear that the *second* pivot array is central for this query. Instead of scanning the data array one element at a time, we can scan the array of pivots that represent the sorting on (d, h). If we annotate the pivot arrays with information about the range of attributes of the data they contain, we will be able to make decisions about entire subsets of contiguous data at once. This is already somewhat useful, but imagine, for example, a natural interactive query in which users are interested in studying the same histogram as before, but for a particular subset of days of the week. As we have currently described Hashedcubes, there is no connection between the different pivot arrays, and so we cannot use information about values in one dimension to speed up queries of a different dimension. But this is easy to fix: after sorting on a finer attribute, we annotate the "coarse pivots" with the range of pivots that they represent in the next finer dimension. In our example, the array of d pivots will be annotated with the boundaries they represent on the array of (d, h) pivots; the (d, h) pivots, in turn, will be annotated with the boundaries they represent in (d, h, p) pivots, and so on. Now consider our working queries above again. In the same way that we exploited the query attribute values to skip entire ranges of data values by scanning the (d, h) pivot array, we can scan the d pivot array to skip entire ranges of the (d, h) pivot array itself. This is the central insight behind Hashedcubes. The astute reader will have undoutedbly noticed that if we instead wanted to filter on network ports, we could not escape a scan of a relatively large (d, h, p) pivot array. This is correct, and we discuss this further in Section 3.4.6.

3.4.2 Construction Algorithm

The algorithm for building Hashedcubes requires an ordering of dataset dimensions (e.g. first spatial, then categorical, and finally temporal). In what follows, we will sometimes use terms like “above” and “below” to refer to precedence relationships in this ordering. Once defined, a linear array called *Hash* is associated with a root pivot $[0, n-1]$, which represents the initial partition containing the universe of n elements. Each element of the *Hash* array is an integer that points to a record in the dataset. The *Hash* array can be stored in a random or sequential ordering. For every dimension of the indexing scheme, each partition (here forth referred as a bin) of each object is indexed using pivots. Bins have different interpretations for each dimension. Bins represent regions for a spatial dimension, specific values or ranges for a categorical dimension, or time intervals for a temporal dimension.In a input array of n elements all entries belong to the same bin, represented by a pivot $[i_0, i_1]$. Each dimension receives as input a list of pivots and outputs a list of pivots. The first dimension receives as input the root pivot. Subsequent dimensions receive the list of pivots created from the previous dimensions. Sorting is performed in each bin to group elements. The bin delimited by a given pivot is further refined as necessary to create subset bins, represented by a new list of pivots. After processing each dimension a new list of pivots is generated. A hierarchy of pivot lists connects the bins created in each dimension.

Hashedcubes supports three distinct dimension types: spatial, categorical and temporal. The pivot hierarchy for these three dimension types can be built in any order. Since a bin at a given dimension is a subset of a bin in the previous dimension, a list of pivots represents subsets for all previously defined dimensions. This allows to remove dimensions from the representation, which is useful for managing memory consumption. The pivot hierarchy mimics a tree hierarchy since each pivot represents a set that can be further divided into a variable number of subset pivots, but notably, it does not store edges from one dimension to another. Sibling pivots (nodes) are stored as lists. Because each dimension stores collections of pivots, and pivot indices are always offsets into the data array, dimensions can be treated independently of each other. This allows the algorithm which executes queries to skip dimensions that are not referred to by in the query. Furthermore, the cardinality of the subset represented by a pivot can be directly obtained from the pivot indices; this way, the size of an aggregation can be directly determined by the list of pivots themselves.

Figure 3.3: A comparison between the computation of Nanocubes and Hashedcubes. Note that Nanocubes pre-compute more aggregations, which tends to lead to lower query times but larger memory consumption. Hashedcubes, in contrast, uses a sparser set of preaggregations in its query execution engine.



(a) Hashedcubes

(b) Nanocubes (adapted from (Lins; Klosowski; Scheidegger, 2013))

Query	Hashedcubes	Nanocubes
Count[<0,1>] **or** Count[<10,01>,<11,01>]	Pre-computed	Pre-computed
Count[all<Android>] **or** Count[all<iPhone>]	Compute On-the-fly	Pre-computed

We use the Figure 3.2 to illustrate different aspects of Hashedcubes. The input data consists of 10 points using the schema [[Latitude, Lonngitude], [Device], [Time]]. In Figure 3.2b step 1, the array is re-ordered along the first level of the quadtree and three partitions are created associated to quadrants 0, 2, and 3 (the quadrants that contain points). Three pivots are created ([0-5], [6-7], [8-9]) to delimit these partitions. In step 2 the array is re-ordered along the second level of the quadtree. Note that only the first quadrant of the quadtree is subdivided in this step, and therefore only the partition affected (associated to the pivot [0..5])) is updated, leading to two new pivots ([0..2] and [3..5]). In steps 3 and 4 the process is similar, but using the categorical and temporal dimensions to create further partitions in the data. In the top of Figure 3.2c we compare the input values of the array to the final re-ordering obtained after successive partitions of the data. In Hashedcubes it suffices to keep the final array along the pivots created at each step to recover the partitions created during these steps. In the bottom of Figure 3.2c we show the list of pivots created at each step and stored by Hashedcubes. The list of pivots correspond to partitions induced in the first and second levels of the quadtree, and the categorical partition, in this case if device used was Windows (W) or Linux (L).

In contrast to other data cube alternatives (GRAY et al., 1997b; Lins; Klosowski; Scheidegger, 2013; LIU; JIANG; HEER, 2013), Hashedcubes does not precompute aggregations across every possible set of dimensions. Instead, it leverages the pivot hierarchy to compute missing pre-aggregations on-the-fly. Consider in Figure 3.3 the problem of computing the number of all objects labeled as *Android* or *iPhone* in the categorical dimension. Hashedcubes does not pre-compute this information. Although this means that such queries will require a scan over a potentially large portion of the array, the fact that Hashedcubes stores these in an array (as opposed to a pointer-based data structure) means that the aggregations can be computed relatively efficiently. In fact, allowing these worst-case scenarios to occur is precisely what is responsible for the low memory consumption in Hashedcubes. The query algorithm is described in Section 3.4.6.

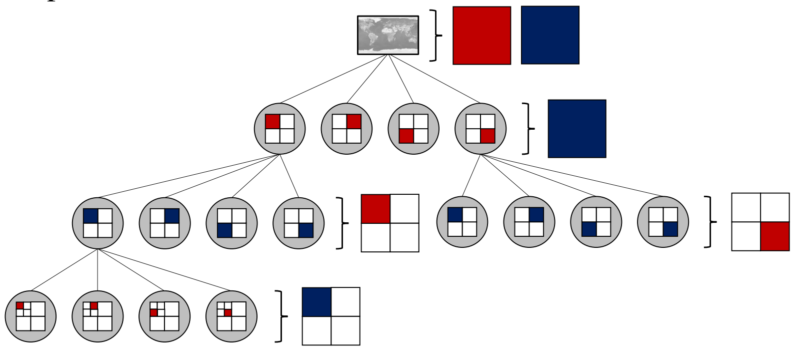
3.4.3 Spatial Dimensions

Efficiently answering queries involving spatial attributes typically requires the use of hierarchical spatial data structures (SAMET, 2005). In Hashedcubes the spatial dimension is represented as a quadtree, a hierarchical data structure often used to represent geo-spatial data where the space is recursively divided into 4 regions (SAMET, 2005). Each quadtree node is associated with a pivot that delimits the objects contained in that quadrant. If a query matches the exact region represented by a node, then the pivot represents the aggregation result for that query. Otherwise, we compute the minimal disjoint set of nodes that cover the query region. We note that during an interactive session, the viewport region of the screen can be interpreted as a spatial query. Although Hashedcubes can process dimensions in any given order, in our experiments we chose to use the spatial dimension first in the ordering of dimensions to increase the speed in which geo-spatial queries can be answered.

The algorithm for building spatial dimensions associates each record within each pivot range to its current quadtree quadrant. Sorting is used to group records belonging to the same region, and consequently, quadtree nodes store the pivot that delimits the records for that specific subdivision. As we mentioned above, the schemas we use typically start with spatial dimensions. Therefore, the input is a single pivot (root) representing the data universe and only a unique quadtree is allocated.

Hashedcubes supports multiple spatial dimensions, but this process is different from single spatial dimensions. Each spatial dimension is associated with a quadtree. Instead of building each spatial dimension sequentially, Hashedcubes interleaves the construction of each quadtree, refining one level of each quadtree at a time. Consider a dataset of phone calls, with two geographical locations, one from the caller and another from the receiver. The root of the quadtree represents all data. At each level of the quadtree the records are subdivided according to the current spatial attribute (e.g. odd and even levels

Figure 3.4: Multiple spatial dimensions. In this example one quadtree is created for each of the two spatial dimensions, red and blue. The quadtrees are used alternately in Hashedcubes to partition the data.



can be associated to origin and destination locations respectively). By using an interleaved quadtree, queries with multiple region constraints are answered by traversing a unique data structure, since quadtree nodes stores the bounding box and the pivot that matches precisely to all aggregates from that regions. Figure 3.4 illustrates this process.

Another important aspect of the Hashedcubes quadtree implementation is the minimum leaf size. Every dimension output is the input for the following dimensions, while each pivot is subsequently refined to represent subsets of specific attributes. Smaller pivots cause the creation of a greater number of subsets. Consider Figure 3.2d. For every input of the *spatial dimension*, it can at most output 2^{2n} subsets, where n is the maximum quadtree subdivision. For every input of *categorical dimension* it can output at most two subsets (*windows* or *linux*). Thus, the output size is directly dependent on the input size. The leaf size is a crucial factor for memory usage and performance of Hashedcubes, and is discussed in Section 3.8.

3.4.4 Categorical Dimensions

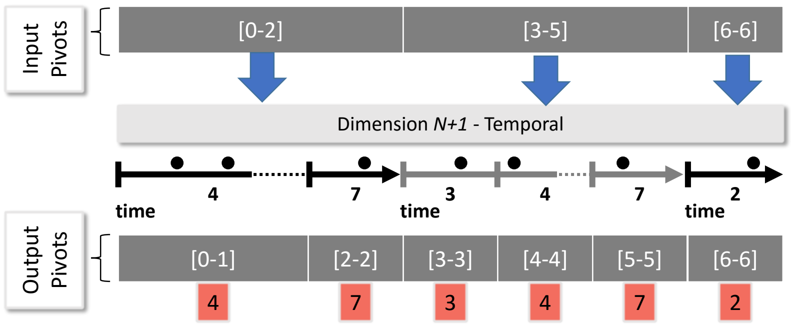
Categorical attributes of multidimensional datasets are usually divided into specific values or ranges. The processing of such attributes in Hashedcubes produces a list of pivots that groups data in bins for each categorical value or range. By varying the granularity of the Hashedcubes query results, categorical queries form the basis for histograms, binned scatterplots and time series plots.

To process a categorical dimension, each record attribute is tagged and a position in the output list of pivots is computed. This algorithm compares an element against all dimension attributes and returns a bin tag. Once this finishes, the sorted list of pivots is

Table 3.1: Subset of queries supported by Hashedcubes HTTP API.

Queries (in natural language)	Spatial	Categorical	Temporal	URL
heatmap of all check-ins in Mondays	drilldown	rollup	rollup	/tile/tile/0/0/0/0/8/where/day_of_week=Mon
hour of day histogram of check-ins in the USA	rollup	drilldown	rollup	/group/hour_of_day/region/0/USA
scatterplot of hour of day/day of week of check-ins	rollup	drilldown	rollup	/scatter/field/hour_of_day/field/day_of_week/region/0/Eu
check-ins in Fridays and between Jan and Feb of 2010	rollup	rollup	drilldown	/tseries/tseries/0/Jan-2010/Feb-2010/where/day_of_week=Fri

Figure 3.5: Temporal dimension indexing. A period of time is represented by a dense list of timestamped pivots. Each black circle represents a record that has been tagged to a specific bin.



created. For a categorical dimension of n distinct values or ranges, at most n pivots can be created. Hashedcubes stores a structure called *CategoricalNode* which implements a dense vector based on the number of unique attributes. Consider the *categorical dimension* in Figure 3.2d, which has as input a list of pivots of size 4. Every input creates a *CategoricalNode* that has a vector with two pivots, representing either *Windows* or *Linux*. The result of processing this dimension creates a list of pivots of size 6, with 4 CategoricalNode objects (*object 1*: [0-0],[1,2]; *object 2*: [3,4],[5,5]; *object 3*: [6-7]; *object 4*: [8-9]).

Unlike the processing of multiple spatial dimensions (which are processed in an interleaving fashion), multiple categorical dimensions are generated in sequence.

3.4.5 Temporal Dimensions

We take advantage of the fact that a pivot represents an interval to represent temporal dimensions. Consider the example of a temporal dimension that needs to be processed to create bins for each different day. The building algorithm classifies each element of the input in the corresponding bin. The result of this process is a sparse list of sets since a bin is created if it has, at least, one record. From this list, a compact list of timestamped pivots is created, as illustrated in Figure 3.5.

The algorithm for building the temporal dimension is similar to the one for categorical dimensions. It tags each record with its respective bin since epoch time. Hashedcubes supports any granularity multiple of milliseconds, and the time interval is defined by the building schema (e.g., 15 minutes, 1 hour, 4 hours, 1 week, etc). Take as an example a schema that aggregates time by the hour, and two records with a difference of 40 minutes. These records are tagged to the same bin, and consequently, represented by a single pivot.

This algorithm enables temporal queries to be efficiently answered without requiring a hierarchical data structure. This is accomplished with two executions of a binary search algorithm, which finds the pivot with the smallest and greatest values from the period of time. This is precisely the same algorithm used by Lins et al.'s Nanocubes (Lins; Klosowski; Scheidegger, 2013).

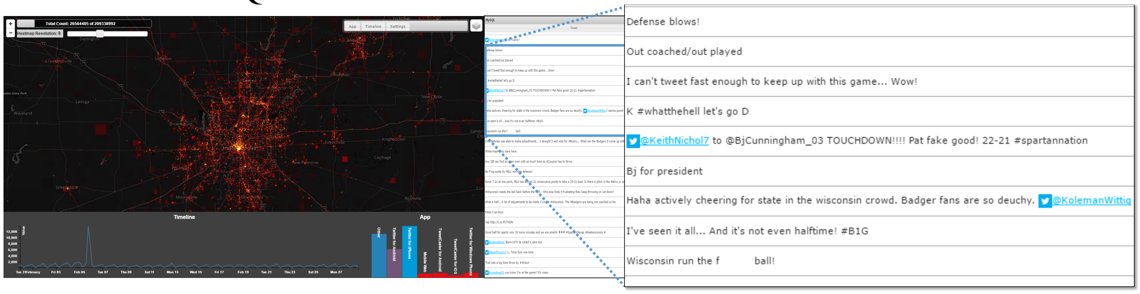
3.4.6 Queries

A query into a Hashedcubes comprises a set of *clauses*. Each clause corresponds uniquely to a dimension, and defines either *constraints* on values or *group-by* directives (often a dimension will contain both a group-by directive and a value constraint). Constraint clauses specify regions of the dataset to be aggregated over, while group-by clauses indicate partition boundaries for the result, in direct analogy to SQL's group by clause (eg. different bins of a monthly histogram as in a "group by month" SQL clause, or nodes of a quadtree for a multiresolution heatmap plot).

The result of a Hashedcubes query is a list of aggregated pivots. As discussed in Section 3.4, Hashedcubes does not store precomputed aggregations across every possible set of dimensions. Instead, it materializes only a portion of all combinations (corresponding to a strict prefix ordering of the dimensions as alluded above). The query execution algorithm takes advantage of the pivot hierarchy to compute the missing aggregations on-the-fly, scanning subintervals of dimensions as necessary.

Most queries contain a group-by clause. In queries broken down by latitude and longitude (as in those which generate heatmaps), the spatial dimension is that clause. In queries broken down by categorical attributes or timestamps, any of the multiple categorical or temporal dimensions can be the group-by clause. Take as example the schema in Figure 3.2d. Assume we are interested in the count of all objects with quadrants 0 and 1 as spatial coordinates and categorical attribute $Windows$. In this case, the result of the query is exactly the contents of a single pivot in that dimension, and no aggregations are necessary. This query is efficient because the constraint clauses form a prefix over the

Figure 3.6: Visual exploration of the twitter dataset during Super Bowl 2012. In addition to enabling real-time exploration using a wide range of visual encodings, with support to brushing & linking in any dimension, Hashedcubes allows the access to the text of tweets from an external SQL server.


Defense blows!

Out coached/out played

I can't tweet fast enough to keep up with this game... Wow!

K #whatthehell let's go D

@KeithNichol7 to @BjCunningham_03 TOUCHDOWN!!!! Pat fake good! 22-21 #spartannation

Bj for president

Haha actively cheering for state in the wisconsin crowd. Badger fans are so deuchy. @KolemanWittig

I've seen it all... And it's not even halftime! #B1G

Wisconsin run the f ball!

ordering of the dimensions (in fact, it's the entire dimension set). Consider, on the other hand, a query that requests the count of all objects with categorical attribute $Windows$, regardless of spatial coordinates. In this case, there is no single pivot storing the final result, and so it is clear that some on-the-fly aggregation will be required.

The full algorithm proceeds as follows. Initially, the query range is the dataset universe represented by the root pivot $[0, n-1]$. The query result in each dimension is a delimiting list of pivots of the selected data, thus, these lists become the new range query, similar to a breadth first search algorithm that uses two lists, one for expanding and one for temporary storage. This process is iteratively repeated until the last dimension. Note that, unlike tree-based data structures, scans happen along arrays. Such approach tend to offer appealing performance, since the CPU cache automatically optimizes burst memory operations (GOODMAN, 1983; KRISHNAMOHAN; FARMWALD; WARE, 1996).

3.5 Implementation

The current implementation of Hashedcubes uses a simple client-server architecture. The server reads the data from a file (e.g. CSV tabular files), builds the data structure and enters an event loop that waits for queries from the client. The server is implemented in C++. Since Hashedcubes uses linear-based memory structures such as sorted arrays, it preallocates chunks of memory to avoid the overhead of repeated memory allocations and deallocations, which are common operations in tree-based data structures. Besides the sorting of the index arrays, Hashedcubes does not require any data precomputation prior to building its data structure. The sorting of the data array dominates the construction time, as we discuss in Section 3.7.2.

For the representation of spatial values, Hashedcubes uses the spherical Mercator projection popular with map tile providers such as OpenStreetMap (HAKLAY; WEBER, 2008). Typically, map tiles providers use coordinates (x, y, z) for each tile image. The tuple $[x, y]$ corresponds to integer addresses, while z represents the zoom level, in most cases varying from 0 (maximum zoom out) to 18 (maximum zoom in). Each zoom increment doubles the $[x, y]$ resolution, and consists of 4^n tiles. We choose to limit the spatial coordinates to a maximum of 26 levels: the maximum zoom value plus 8, corresponding to the typical tile size of 256x256. The 26-level subdivision naturally yields a 26-bit address for each of the x and y coordinates, and these addresses can be easily employed for the hierarchical sorting in spatial coordinates.

The server is easily parallelizable since the data structure does not change after building. It exposes the querying API via HTTP (as in Table 3.1) through a web service implementation that handles concurrent requests in multiple threads. In the front-end, the prototype client is written in Javascript, SVG, and HTML5; notable libraries include D3 (BOSTOCK, 2015) and Leaflet (AGAFONKIN, 2014), as shown in Figure 3.6.

3.6 Datasets and Schemas

In this section, we report an evaluation of Hashedcubes using a collection of publicly-available datasets. We collected seven datasets that range from 4.7 million to 1 billion records, including some used in other data cube visualization proposals, as well as the schema they used. In addition, we introduced some variations on the schemata used in previous experiments in order to properly stress the features of both Hashedcubes and previous systems. We summarize all of the schema variations and datasets in Table 3.2.

3.6.1 Location-Based Social Networks

Brightkite and Gowalla are two former location-based social networks: users participated by sharing their locations via check-ins events. Both datasets are publicly available in Leskovec's *Stanford Large Network Dataset Collection* (LESKOVEC; KREVL, 2014). They consist of time and location information of user check-ins, collected by Cho et al. (CHO; MYERS; LESKOVEC, 2011a). Brightkite check-ins range from April 2008 to October 2010, and Gowalla from February 2009 to October 2010. We built Hashedcubes using two different schemas for these datasets. The first one replicates the schema used by Nanocubes and encodes latitude and longitude as spatial information, hour of the day and day of the week as categorical variables, and check-in time as temporal variables. The second one replicates the imMens schema and encodes latitude and longitude as spatial information, hour of the day, and day of the month as categorical information. In Figure 3.1, we use Hashedcubes to visualize Brightkite check-ins in Europe and to highlight Brightkite releases of its iOS app and its 2.0 platform version.

3.6.2 Airline On-Time Performance

The U.S. Department of Transportation tracks the on-time performance of domestic flights by U.S. air carriers. This dataset was made publicly available in (American Statistical Association Data Expo., 2009; WICKHAM, 2011), and covers over 121 million flights in a 20 year period, from 1987 to 2008. Records include over 29 fields. We used three different schemas for this dataset. The first one encodes the origin airport as spatial information, departure delay and carrier delay as categorical information, and departure delay as temporal information. This is the same schema used in Nanocubes. The second schema is the one used by imMens, and encodes only categorical information. The day of the week, year, carrier, arrival delay and departure delay are the categorical information. Note that the arrival delay and departure delay are encoded as 15 minutes interval bins, and were designed to be visualized using a scatter plot. The last schema is designed to exploit the Hashedcubes ability to work with multiple spatial dimensions, so we encoded origin and destination airports as spatial information.

3.6.3 SPLOM

The ScatterPlot Matrix (SPLOM) benchmark (KANDEL et al., 2012) was designed to stress test the data cube technology, and has been used as validation in recent big data visualization proposals (LIU; JIANG; HEER, 2013; Lins; Klosowski; Scheidegger, 2013). It consists of a collection of synthetic elements with up to five dimensions. The first, second and fifth dimensions are independent and normally distributed. The third and fourth dimensions are, respectively, linearly and log-linearly dependent with the first. As a synthetic dataset, we used five different bin sizes per dimension, from 10 to

50, and varied the elements from 100 million up to 1 billion to stress test Hashedcubes (Figure 3.7a).

3.6.4 Twitter

The data consists of geolocated tweets collected from the (formerly open) Twitter API between November 2011 and June 2012 that originated in the United States. We used two different schemas, namely twitter-small and twitter. The first one encodes the record origin as spatial information, device used as categorical information, and record collection time as temporal information. The second schema adds the application and language, respectively 4 and 15 distinct values, as categorical informations. In Figure 3.1 we present and overview of tweets in USA, and a close-up in the date and region of Superbowl 2012.

3.6.5 NYC Yellow and Green Taxis

The NYC Taxi and Limousine Commission (TLC) collects and provides monthly trips records from yellow and green taxis from New York City. Records include over 21 fields that capture pick-up and drop-off times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, driver-reported passenger counts, and others. While yellow taxis are able to pick-up passengers in any of the five NYC boroughs, green taxis are only allowed to pick-up passengers in outer boroughs and in Manhattan above East 96th and West 110th Streets. For each dataset, we used two different schemas, both encoding pick-up and drop-off locations as spatial information. The first schema encodes time as week bins along with categorical information: day of the week and hour of the day. The second schema encodes time as hour bins. In Figure 3.1, we highlight the use of Hashedcubes to analyze pick-up locations from the green taxis dataset.

3.7 Performance Results

In this Section we discuss the performance results of Hashedcubes. We compare the Hashedcubes memory usage, construction and query time to recent data cube visualization proposals, namely Nanocubes (Lins; Klosowski; Scheidegger, 2013) and

Table 3.2: Overall summary of the relevant information for building Hashedcubes.

dataset	objects (N)	leaf-size	memory	time	pivots (P)	schema
splom-10[1,2]	1.0 B	N/A	5 MB	38:32 m	26 K	d1 (10), d2 (10), d3 (10), d4 (10), d5 (10)
splom-50[1,2]	1.0 B	N/A	349 MB	46:28 m	12.7 M	d1 (50), d2 (50), d3 (50), d4 (50), d5 (50)
brightkite[1]	4.5 M	32	366 MB	7 s	6.7 M	lat0, lon0, hour of day (24), day of week (7), time (week)
brightkite[2]	4.5 M	32	375 MB	10 s	6.8 M	lat0, lon0, month of year (12), hour of day (24), day of month (31)
brightkite-alternative	4.5 M	32	468 MB	8 s	8.0 M	lat0, lon0, time (week), hour of day (24), day of week (7)
gowalla[1]	6.4 M	32	743 MB	13 s	12.6 M	lat0, lon0, hour of day (24), day of week (7), time (week)
flights	121.2 M	32	1.5 GB	06:55 m	61.0 M	lat0, lon0, lat1, lon1, departure delay (9), carrier (29), time (4 hours)
flights[1]	121.2 M	32	457 MB	03:56 m	19.5 M	lat0, lon0, departure delay (9), carrier (29), time (4 hours)
flights[2]	50.3 M	N/A	18 MB	12 s	396 K	day of week (7), year (21), carrier (29), arr_delay (174), dep_delay (174)
twitter-small[1]	210.6 M	64	4.9 GB	10:53 m	137 M	lat0, lon0, device (5), time (4 hours)
twitter[1]	210.6 M	64	9.4 GB	12:04 m	203 M	lat0, lon0, app (4), device (5), language (15), time (4 hours)
green-taxis-small	24.5 M	64	788 MB	01:35 m	27 M	lat0, lon0, lat1, lon1, time (hour)
green taxis	24.5 M	64	3.0 GB	01:49 m	52 M	lat0, lon0, lat1, lon1, day of week (7), hour of day (24), time (week)
yellow-taxis-small	224.1 M	64	7.0 GB	18:14 m	243 M	lat0, lon0, lat1, lon1, time (hour)
yellow-taxis	224.1 M	64	12.6 GB	20:38 m	473 M	lat0, lon0, lat1, lon1, day of week (7), hour of day (24), time (week)

[1]Schema used by Nanocubes. [2]Schema used by imMens.

imMens (LIU; JIANG; HEER, 2013). Table 3.2 summarizes benchmark results for all schema variations and datasets. The number of records (N) in the dataset, quadtree leaf-size, memory usage, time to build and the accumulated number of pivots (P) across all data structure are reported.

3.7.1 Memory Usage

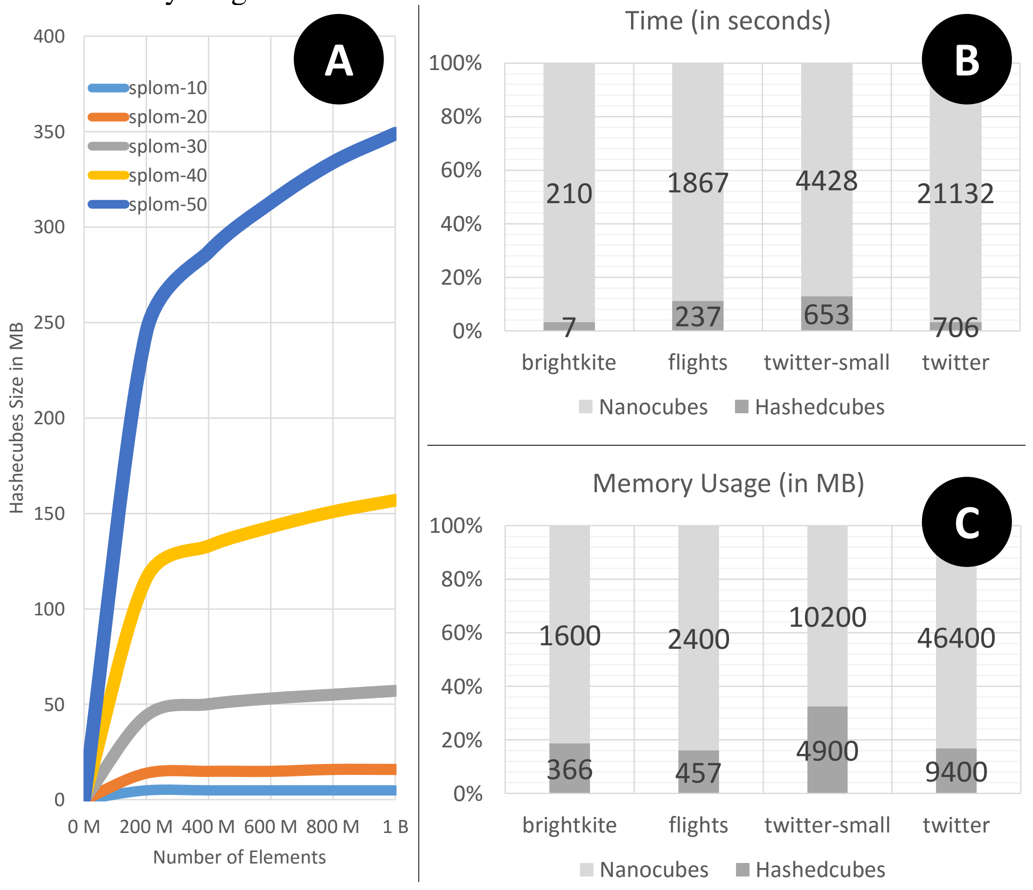
Memory usage in Hashedcubes is directly proportional to the number of pivots, i.e., the number of used bins per dimension. Figure 3.7a shows the memory growth for the SPLOM dataset ranging from zero to one billion inserted records. We used five schema variations that range bin size from ten to fifty in each dimension. Records from this dataset are collected from synthetic generators that have a normal distribution, which means that the set of high probability values are quickly sampled, making harder for new records with an unseen bin. It highlights an effect known as *key saturation*. Due to the key saturation effect, most inserted records does not require additional memory since their pivots were already present in the Hashedcubes index, a phenomenon that performs an important role to reduce memory requirements.

When comparing Hashedcubes to recent data cube strategies, memory usage sees a breakthrough from current state-of-the-art data cube proposals, enabling the visualization of a much larger set of scales and more complex schema configurations than imMens and Nanocubes. Compared to Nanocubes, we find a reduction factor of up to 5.2x in the best case, as shown in Figure 3.7c. Building the Hashedcubes for brightkite, flights, twitter-small and twitter schemas, requires 366MB, 457MB, 4GB and 9.4GB of memory, respectively. For the same schemas, Nanocubes requires 1.6GB, 2.3GB, 10.2GB and

46.4GB, enough for present day servers, but above that of typical notebooks and workstations. imMens uses a dense indexing to speed up aggregation time and to simplify parallel query processing, but this implies that memory usage is proportional to the cardinality of its key space. Furthermore, it lacks support for compound brushing of more than four dimensions, once it requires computing prohibitively large 5-dimensional data tiles for the adopted approach.

We also evaluated Hashedcubes for schemas with multiple spatial dimensions, a feature that was not supported by Nanocubes and imMens in their initial public releases. For that, we introduced schema variations and two unstudied datasets, namely, the green and yellow NYC taxis. These datasets are particularly hard because both have a very restrict spatial region, thus pushing spatial dimensions data structures to deeper levels of subdivision. Moreover, we tested two time resolutions, by hour and over a week along with day of week and hour of day categorical attributes. We have attempted to create Nanocubes for these schemas, but found them to take a prohibitively large amount of memory. Before killing the nanocube process, we estimated the eventual memory usage of

Figure 3.7: **(a)** Hashedcubes memory usage growth while inserting SPLOM dataset elements. Notice the *key saturation* effect. **(b)** and **(c)** compare Hashedcubes construction time and memory usage to Nanocubes.



the yellow-taxis-small schema to be around 124GB for a pair of 20-bit quadtree addresses, and 321GB for 25-bit addresses (and an estimated five hours of construction time). We made no attempt to generate a nanocube of the full yellow taxis schema.

3.7.2 Construction Time

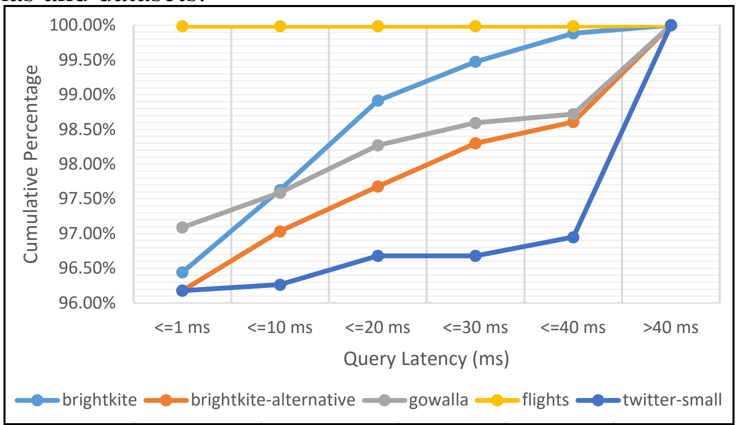
Construction time was a relevant factor when designing Hashedcubes. The construction algorithm was optimized for speed by avoiding repeated memory allocations and deallocations. The bottleneck of this algorithm are the sorting phases, specially when handling spatial dimensions. The pivot hierarchy uses a sorting step for every quadtree, which can be very demanding for datasets with restricted geographical coverage and multiple spatial dimensions, since these cases tend to generate trees next to the maximum recursion depth supported. Compared to Nanocubes, we obtained a reduction factor of up to 30x in the best case, as shown in Figure 3.7b. On average, the construction time is about 10 times faster.

3.7.3 Query Time

We used a set of real-world queries graciously provided by AT&T Research to assess query latency. Query requests were collected on the public Nanocubes (Lins; Klosowski; Scheidegger, 2013) web site, in which users performed brushing and linking across dimensions of Brightkite, Gowalla, Flights and Twitter datasets. This set provides a sample of common actions when exploring real-time interactive systems using a wide range of visual encodings. Unlike synthetic benchmarks, it allows to validate Hashedcubes in an uncontrolled environment. We implemented a script that translates Nanocube queries to Hashedcubes queries and compares the results of both proposals. For that, we used the same schema from Nanocubes.

Figure 3.8 shows the percentages when the set of queries is executed in an Intel Core i7 4790 CPU. We report the median, mode, mean, standard deviation and maximum latency for each of the tested schemas. Typically, Hashedcubes performance level is within the real-time budget (<40ms *or* >25fps); only one in fifty queries takes more than 40ms. The most time-consuming queries are those which require a large number of aggregations of many small pivots. These typically happen when the query constraints

Figure 3.8: Cumulative percentages of query latency from real-world scenarios. The vast majority of queries are answered within the real-time budget ($<$40ms *or* $>$25fps) for different schemas and datasets.



statistic/ dataset	brightkite	brightkite alt.	gowalla	flights	twitter-small
queries N	507880	507880	102430	215980	48190
median	0 ms	0 ms	0 ms	0 ms	0 ms
mode	0 ms	0 ms	0 ms	0 ms	0 ms
mean	0 ms	1 ms	1 ms	0 ms	4 ms
stdev	4.21 ms	11.02 ms	7.19 ms	1.03 ms	66.93 ms
maximum	94 ms	281 ms	114 ms	159 ms	1382 ms

are specified over a variable that has been "finely split" over a large range of indices, and yet no filtering in previous dimension rejects has occurred. In the worst case, this might degenerate to a linear scan over the dataset. For other schemas and datasets, Hashedcubes presented similar frequency distribution, consistently answering many queries under $40ms$ for various rollups and drill down test combinations. The server to client latency was dominated by transference of geographical tiles information.

Nanocubes have a very small worst-case value, around 12ms. imMens sustains a 20ms update time on average. It has to be noted, however, that both solutions uses pre-computation and a higher memory footprint in favor of faster queries. Hashedcubes, instead, balances these two variables and allows the real-time exploration and analysis of datasets that previously required a prohibitory amount of space. Moreover, it supports more flexible schema configurations that enables re-ordering and multiple spatial, categorical and temporal dimensions.

3.8 Discussion

The underlying concept behind Hashedcubes, the pivot hierarchy, can be constructed in any given order. In addition, it allows a natural integration with an external database to complement visual queries. In this Section, we discuss these two extensions and how the quadtree leaf size impacts memory usage and visual accuracy.

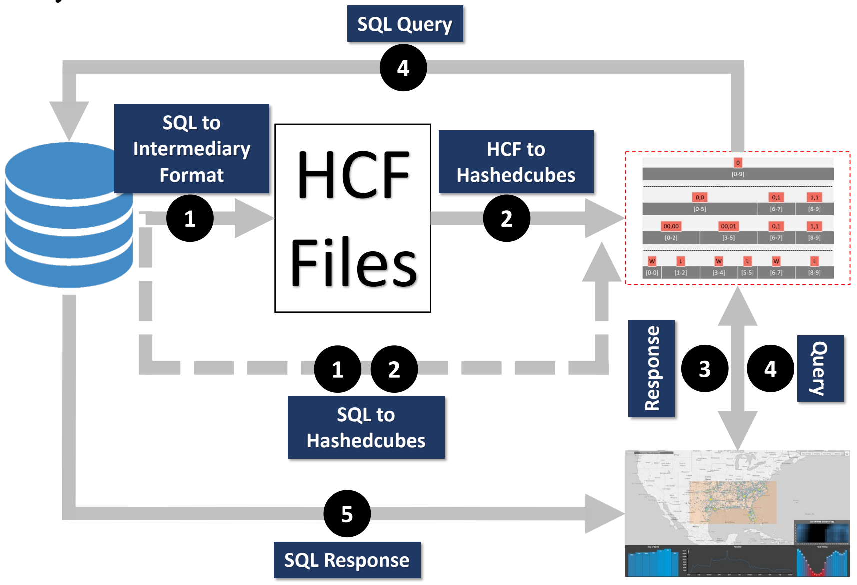
Exchanging the Pivoting Order. Exchanging the order in which the variables are sorted impacts both memory usage and running time of specific queries. In Figure 3.8 we compare two schemas of the same dataset: *brightkite* and *brightkite-alternative*. The set of real-world queries described in Section 3.7 was used to test the Hashedcubes implementation. The alternative schema, using a spatial-temporal-categorical ordering, notably increases both standard deviation and maximum query time from 4.21ms and 94ms to, respectively, 11.02ms and 281ms. Moreover, it increases memory consumption by 25%. On the other hand, in this schema temporal queries answer much faster since there are fewer pivots that need to be processed by the querying algorithm. Such tradeoffs can be considered by a database administrator to choose one layout over another. Automatically tuning the ordering of variables, or possibly creating redundant Hashedcubes instances to process different queries, is a natural area for future research.

Integration with Database of Record. Large data visualization systems like imMens and Nanocubes, along with Hashedcubes, can be considered *approximate databases*, which means that they use data aggregation which might discard some information of the original record. The underlying concept behind Hashedcubes allows a simple integration with external databases. The retrieval of complementary information can be useful, for example, when datasets have text attributes along with spatial, categorical and temporal values, or when these values are not relevant for the exploratory interactive system itself. All real-world datasets used to validate Hashedcubes contain additional information that is ignored by the schema configurations. In Figure 3.6 we show the visual exploration of a large dataset associated with the retrieval of complementary data from an external SQL server.

Hashedcubes allows to recover original data by associating the pivot indexes with an external index, for instance, an SQL index. As shown in Figure 3.9, data is loaded from our intermediary binary format (to obtain faster building times) or directly from the SQL server, and sorted out accordingly to the external ordering. Hashedcubes answers queries in real-time and simultaneously triggers asynchronous SQL queries based on the pivot selection. This natural extension encourages the complement of visual queries with external information.

Leaf-Size Trade-off vs Visual Accuracy. During the construction of Hashedcubes, the output of every dimension serves as input for the following dimension, and each pivot is

Figure 3.9: Hashedcubes supports recovering the original data by using a linking structure. Pivots represent the values from the SQL index, which allows to efficiently match all rows of a given query. Hashedcubes can be built directly from a SQL database or from an intermediary format.



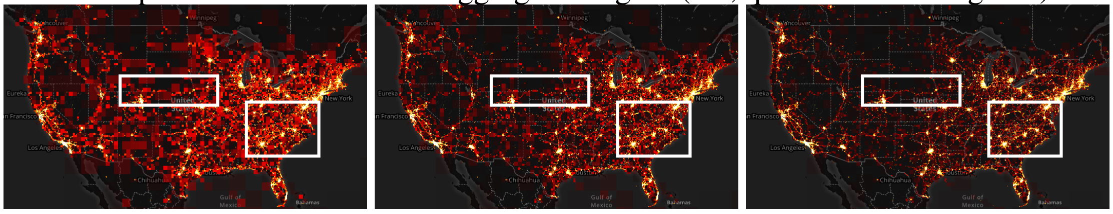
subsequently refined to represent smaller data subsets. Spatial dimensions adopt a minimum quadtree leaf-size to balance running time, memory usage and visual accuracy, as shown in Figure 3.10 (a), (b) and (c). The leaf-size threshold creates a phenomenon called *truncated pivot*. This indicates that a given spatial region will be no longer subdivided if a minimum leaf-size is reached. Since visual accuracy was a relevant factor when designing Hashedcubes, we implemented a specific heatmap visualization that allows identifying truncated pivot occurrences (Figure 3.10a).

Truncated pivots are typically found in smaller geographical regions with very low data sampling, an arrangement which might mask outliers. As a workaround to this issue, Hashedcubes users can integrate external databases to recover precise spatial information of a specific region, as discussed previously. It has to be noted, however, that Hashedcubes supports any leaf-size threshold. The default values for the schemas in Table 3.2 were chosen to achieve a good balance between running time and memory usage while producing a similar visual result when compared to the other data cube visualization proposals (Figure 3.11).

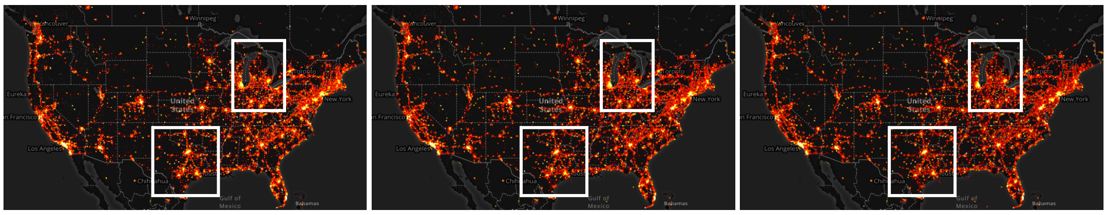
3.9 Conclusions and Future Work

In this paper, we presented Hashedcubes, a fast, easy to implement and memory efficient data structure to answer queries from interactive visualization tools that explore

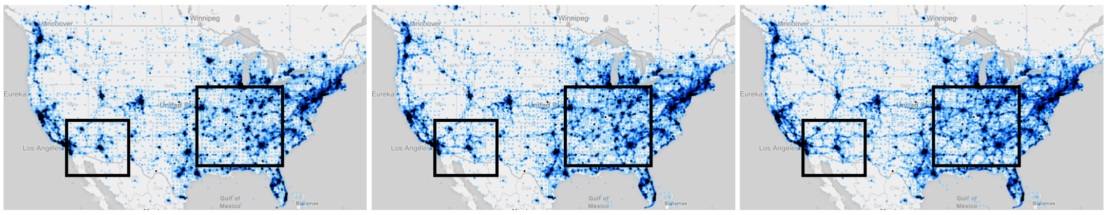
Figure 3.10: Hashedcubes different heatmap visualizations showcase. Notice the leaf size variation from 32 to 8 by looking into the highlighted regions. It impacts running time, memory usage and visual accuracy. **(a)** allows to identify truncated pivot occurrences by representing them as rectangles. Color is a factor of area and occupancy. **(b)** and **(c)** use circles to represent the center of an aggregated region (i.e., quadtree bounding box).



(a) Brightkite overview. Primitive: rectangles, Colormap: red-yellow-white, Density Aware.

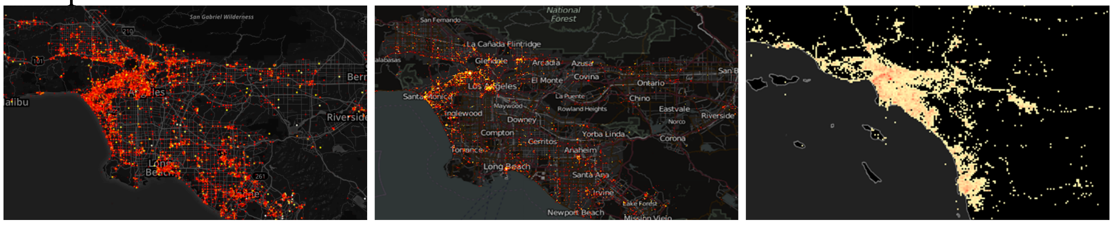


(b) Brightkite overview. Primitive: circles, Colormap: red-yellow-white, Density Aware.



(c) Brightkite overview. Primitive: circles, Colormap: light-blue-dark, Not Density Aware.

Figure 3.11: Los Angeles (United States) city view of detailed Brightkite heatmaps from recent data cube visualization proposals. Apart from the use of different colormaps across Hashedcubes, Nanocubes and imMens, what produces a slightly dissimilar visual appearance, Hashedcubes pivot concept enables a high visual accuracy along with reduced memory consumption when compared against other data cube visualization proposals. Notice that Hashedcubes matches Nanocubes visual representation, even though the latter does not experience leaf-size trade-offs.


Hashedcubes (Circles, Density Aware, Leaf-size: 32)
Nanocubes
imMens (maximum supported zoom by public demo)

and analyzes large multidimensional datasets. Pivot hierarchy, the underlying concept behind Hashedcubes, enables traversal in any order and allows to include multiple spatial dimensions, which is useful to visualize origin-destinations datasets. Furthermore, it supports access to the original data by integrating the data structure with an external database.

Our major contributions have shown that (i) is possible to represent hierarchical and flat data structures using an optimized pivot schema that is stored in a linear fashion way, and (ii) demonstrated that this leads to memory savings over other data cube visualization proposals, as shown in Section 3.7. Taking advantage of the performance level given by Hashedcubes, researchers can develop richer and seamless interactive visualization tools. Moreover, it enables the visual exploration of datasets and schemas that previously take a prohibitory amount of space or time.

As future work, we would like to expand pivot hierarchy concept to automatically find optimal pivoting ordering by calculating a metric that balances running time and memory usage. Since Hashedcubes building algorithms mainly require careful sorting operations that can be adopted to current Web technologies, we also want to explore an exclusively browser-side implementation. Hashedcubes uses a querying algorithm similar to a breadth-first search, with two working lists, one for expanding and another for temporary storage. We envision an alternative approach that use just one list, but that require significant enhancements to the data structure and are left for future work. Another promising research area is the handle of dynamic datasets or streaming data. Hashedcubes can benefit from existing approaches like *Packed-Memory Arrays* (BENDER; HU, 2007b), a concept that aligns surprisingly well with Hashedcubes pivot notion and its worth to be further investigated. Hashedcubes is available as open source software at <https://github.com/cicerolp/hashedcubes>.

4 REAL-TIME EXPLORATION OF LARGE SPATIOTEMPORAL DATASETS BASED ON ORDER STATISTICS

Published in: IEEE Transactions on Visualization and Computer Graphics (TVCG), 2019.
DOI: 10.1109/TVCG.2019.2914446
Qualis CAPES: A1
Authors: Cícero A. L. Pahins, Nivan Ferreira, and João L. D. Comba.

4.1 Abstract

In recent years sophisticated data structures based on datacubes have been proposed to perform interactive visual exploration of large datasets. While powerful, these approaches overlook the important fact that aggregations used to produce datacubes do not represent the actual distribution of the data being analyzed. As a result, these methods might produce biased results as well as hide important features in the data. In this paper, we introduce the Quantile Data Structure (QDS) that bridges this gap by supporting interactive visual exploration based on order statistics. To achieve this, QDS makes use of an efficient non-parametric distribution approximation scheme called p-digest and employs a novel datacube indexing scheme that reduces the memory usage of previous datacube methods. This enables interactive slicing and dicing while accurately approximating the distribution of quantitative variables of interest. We present two case studies that illustrate the ability of QDS to not only build order statistics based visualizations interactively but also to perform event detection on very large datasets. Finally, we present extensive experimental results that validate the effectiveness of QDS regarding memory usage and accuracy in the approximation of order statistics for real-world datasets.

4.2 Introduction

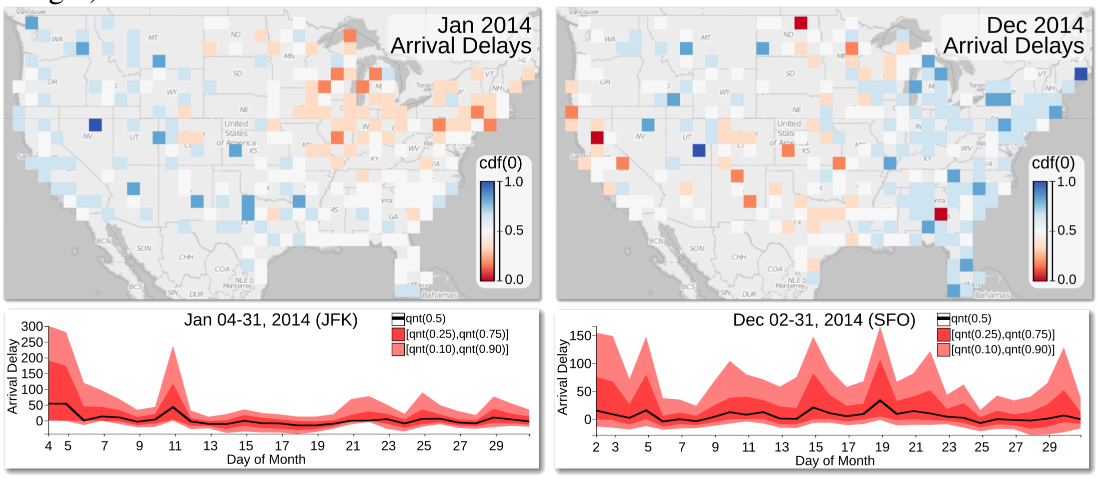
A fundamental problem in modern visual data analysis is how to build data exploration environments that support interactive exploration of large datasets.

This problem has two opposing facets. From one side, the ever-growing complexity and size of datasets bring the need to provide complex navigation and visual summarization capabilities. On the other hand, human perception and cognition pose a challenge on how long the data handling and rendering loop can take. Even small delays on the scale of half a second can have a significant negative impact on the visual data exploration process (LIU; HEER, 2014b). Unfortunately, the ability to produce compelling visual summaries, interaction mechanisms, and interfaces has surpassed our capabilities to create techniques that support real-time data processing for visualization (BATTLE; CHANG; STONEBRAKER, 2016). As a result, there are limitations on the analysis that one can hope to perform interactively. In this paper, we are concerned with the scenario of performing real-time analysis (*i.e.*, virtually immediate results) of large static datasets.

Recent efforts propose sophisticated implementations of precomputed indices (LIU; JIANG; HEER, 2013; Lins; Klosowski; Scheidegger, 2013; Pahins et al., 2017) that store aggregations of a given dataset as solutions to this problem. One limitation of these approaches is the fact that they do not take into account the inherent *distribution uncertainty* due to aggregation: datasets with equal mean and covariance, but with entirely different underlying distributions. Examples of this issue can be seen in the classical Anscombe's Quartet datasets and the work of Matejka et al. (MATEJKA; FITZMAURICE, 2017). The state-of-the-art method Gaussian Cubes (GC) (Wang et al., 2017) supports interactive data modeling by describing the data distribution using parametric Gaussian distributions. Unfortunately, this approach has two drawbacks. First, it relies on non-robust statistics (mean and covariances), i.e., they can be easily affected by outliers. Second, and most importantly, one can not assume real-world data to be normal, and assuming normality can hide essential features of the data.

Contribuitions. To overcome these drawbacks we propose Quantile Data Structure (QDS): a novel data structure that encodes data distributions based on robust statistics while providing support for interactive visual exploration of large spatiotemporal datasets. To achieve this, QDS couples a non-parametric distribution modeling technique called p-digest, based on the t-digest *quantile sketch* (DUNNING; ERTL, 2014) (Sec. 4.5), with a novel indexing structure that reduces the large memory footprint common to datacube structures and enables real-time slicing and dicing. QDS (described in Sec. 4.6) extends the querying abilities of previous approaches by supporting queries with order statistics related aggregations such as quantiles and cumulative distribution. We used QDS in a prototype visual analytics system to demonstrate that these queries provide a powerful tool to interactively build widely used visualizations (such as box plots, equi-depth histograms, and band plots), create new ones (such as the heatmaps based on quantiles and cumulative

Figure 4.1: Analyzing the distribution of flight arrival delays for U.S. airports using QDS. We observe two maps showing the probability of flights being late for January and December 2014. Airports are colored using a divergent color scale representing the cumulative distribution function of the arrival delays at the value 0. We assign red color shades to airports with a higher probability of having late arriving flights and blue shades for airports in which flights are more likely to be early. Notice how the trend changes from more likely delayed flights on the Northeastern airports in January to Southwestern ones (particularly in California) in December. The pattern of delay in January 2014 is due to the snowstorms that pounded the Northeast of the U.S. in January. The Western delay pattern in December is due to the so-called "California's storm of the decade" that affected the region in the middle of December 2014. The temporal band plots on the bottom show the evolution of the arrival delay quantiles (0.1,0.25,0.5,0.75,0.9) for both the JFK (left) and SFO airport (right). Dates with a substantial increase in the median arrival delays (black line) are the peaks of these events (e.g., January 4 on the left and December 15 and 19 on the right).



distribution) (Sec. 4.7) and to perform interactive event detection (Sec. 4.8). Fig. 4.1 illustrates interesting spatiotemporal patterns in the distribution of flight arrival delays for U.S. airports found using QDS. Finally, we provide extensive experimental results (Sec. 4.9) that show the effectiveness of our method for the analysis of real-world datasets scenarios.

4.3 Related Work

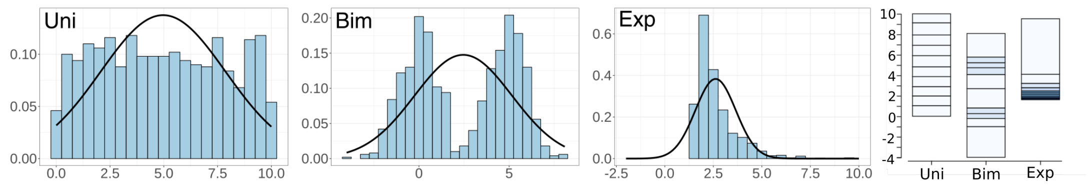
In this section, we review related research on different aspects that play an essential part in this work.

Visualization of Data Distributions. The visualization of statistical summaries is at the core of visual data analysis and visual data communication (POTTER et al., 2010; MACIEJEWSKI et al., 2013). The most common approach relies on visualizations of the mean and standard deviation such as bar charts and error bar plots. This approach is dubious, sensitive to outliers and may not only introduce bias but also hide essential features of the data (as illustrated in Fig. 4.2). For these reasons, this approach has been discouraged by researches in the fields of visualization (CORRELL; GLEICHER, 2014), neuroscience (ROUSSELET; FOXE; BOLAM, 2016) and biology (WEISSGERBER et al., 2015). Scientific publications also incentive the use of more accurate distribution representation such as boxplots (WICKHAM; STRYJEWSKI, 2011) to summarize large datasets (KICK. . . , 2014). Furthermore, recent studies by Kay et al. (KAY et al., 2016) and Fernandes et al. (FERNANDES et al., 2018) showed that presenting detailed distribution information improves decision making compared to scenarios where this information is not present. In addition, these studies showed that specialized visual summaries based on order statistics improved decision making in an uncertainty judgment in a transit scenario. Our work builds on these observations and proposes a data structure that provides accurate distribution approximations for large spatiotemporal datasets.

Interactive Visualization of Large Datasets. The problem of providing interactive analytics and visualization for large datasets has attracted the attention of researchers both in the visualization and databases community. Solutions to this problem follow two main strategies: sampling and pre-computation. The sampling strategy uses progressively increasing samples of a population to approximate/estimate the result of a given query (FISHER et al., 2012a). The survey by Chaudhuri et al. (CHAUDHURI; DING; KANDULA, 2017) describes several techniques for query estimation and data handling in this scenario. In systems using the sampling strategy, users face evolving visualizations that indicate current estimates and, possibly, the uncertainty inherent to the estimation process (JO et al., 2017). While flexible compared to the precomputation strategy, the understanding of the user experience in this scenario is still incipient (MORITZ; FISHER, 2017), thus motivating new visualizations and interactions to support users in analytical environments (FERREIRA; FISHER; KONIG, 2014; MORITZ et al., 2017).

On the other hand, the pre-computation strategy relies on computing aggregations over several dimensions following the datacube concept. Systems such as Immens (LIU; JIANG; HEER, 2013), Nanocubes (NC) (Lins; Klosowski; Scheidegger, 2013) and Hashedcubes (HC) (Pahins et al., 2017) were proposed to reduce the huge memory footprint, but are limited to provide results in counting queries. Recent systems such as TopKube (Miranda et al., 2018) and Gaussian Cubes (GC) (Wang et al., 2017) extend ordinary datacubes to perform more complex analysis in real-time while respecting reasonable memory constraints. QDS also follows the datacube approach. However, we

Figure 4.2: Gaussian distributions are the most common approach of modeling data for analysis and visualization. While this method has theoretical advantages, real-world data is rarely normally distributed. As we observe in (a)-(c) modeling data with normal distributions (black curves) can introduce biases and hide essential features such as multimodality and skewness. As illustrated by the equi-depth histograms produced using p-digest in (d) (darker shades of blue represent higher data density) can efficiently describe the distributions of the other plots.



(a) Uniform Distribution (b) Bimodal Distribution (c) Exponential Distribution (d) p-digest

relax the requirements of exact representation from previous systems to provide a non-parametric approximation of the data distribution. A recent work by Peng et al. (PENG et al., 2018) proposed a hybrid approach that mixes the sampling and precomputation strategies. However, neither this work or the ones cited above support the quantile queries provided by QDS.

Applications of Event Detection in Visual Analytics. Statistical techniques can be used to identify events or anomaly situations, which has been shown to be a powerful tool for visual anlytics (DORAISWAMY et al., 2014). Maciejewski et al. (MACIEJEWSKI et al., 2010) couple visual exploration with modeling strategies to find abnormal spatiotemporal hotspots. Wilkinson et al. (WILKINSON, 2018) use a statistical algorithm for detecting multidimensional outliers. QDS provides a powerful and flexible way to find relevant and complex events using quantiles from the distributions of large datasets.

4.4 Background

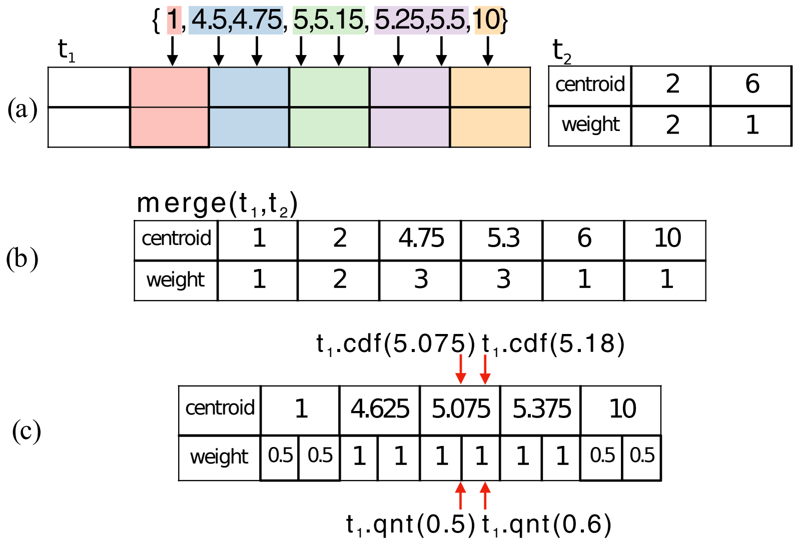
We briefly discuss the background of probability theory and data sketches, and refer to Rosenthal (ROSENTHAL, 2006) and Cormode et al. (CORMODE et al., 2012) for a detailed description. We define the cumulative distribution function (*cdf*) of a random variable X by $F_X(t) = Pr(X \leq t)$. Quantiles are landmark values of a given *cdf* that define specific points where F_X has accumulated a fraction of its total probability. For example, a value t is the q^{th} quantile of F_x if $F_X(t) = q$. Intuitively, one can obtain the value of the q^{th} quantile by $F_X^{-1}(q)$ by simply inverting the *cdf*. In this presentation, we focus on the intuition and overlook the fact that *cdf*'s are not necessarily invertible. We define the first (q_1), second (q_2) and third (q_3) *quartiles* as the quantiles that divide the density in four equal parts, *i.e.*, 0.25^{th}, 0.5^{th} and 0.75^{th} respectively. We define a *random field* as a function F_M that associates to each point in a spatial domain (*e.g.* geographical coordinates) a random variable. We define quantile heatmaps and outlierness queries supported by QDS (Sec. 4.6.1) using random fields.

Unlike moment statistics, such as average and variance, quantiles are robust to the presence of outliers (WILKINSON, 2018). However, it is not possible to combine quantiles of different datasets (e.g. *cdfs*) without processing the input datasets entirely. This limits the use of quantiles in scenarios that require hierarchical/dynamic aggregation such as datacubes. An alternative is to use approximation schemes called *quantile sketches* (PHILLIPS, 2016). A data sketch is “a data structure that can be easily updated with new or modified data and supports a set of queries whose results approximate queries on the full dataset" (PHILLIPS, 2016). Quantile sketches are data sketches that support queries of quantile and *cdf* estimation. Methods vary in memory usage and approximation performance, leading to two groups of methods. The first one has sketches that have proven approximation bounds such as the proposals of Shrivastava et al. (SHRIVASTAVA et al., 2004), Agarwal et al.(AGARWAL et al., 2013a), Karnin et al. (KARNIN; LANG; LIBERTY, 2016) and Felber and Ostrovsky (FELBER; OSTROVSKY, 2017). Such methods have performance requirements which incur in complex algorithms that use large amounts of memory in practice (see discussion in (BEN-HAIM; TOM-TOV, 2010)). The second group of methods lack rigorous algorithmic analysis but relies on heuristics to provide empirical results for query accuracy and reduced memory usage. Examples of methods in this group are the GK sketch (GREENWALD; KHANNA, 2001), the S-Hist sketch (BEN-HAIM; TOM-TOV, 2010) and the t-digest by Dunning (DUNNING; ERTL, 2014).

4.5 The t-digest data sketch

The simplicity and approximation accuracy of t-digest singles it out from other quantile sketches. The t-digest summarizes the (empirical) *cdf* of an input dataset by a set of weighted values called centroids (Fig. 4.3). To choose centroids we group elements on subsequences of varying size following an adaptive strategy. Given an input *compression* parameter δ that defines the maximum number of centroids, the strategy gives high priority to extreme quantiles (closer to 0 and 1), as defined by the function

Figure 4.3: The t-digest sketch: (a) construction of a t-digest t_1. The *cdf* of the input dataset is represented by a set of weighted centroids. (b) Different quantile sketches $t1$ and $t2$ can be combined using the merge operation. (c) A quantile query, qnt(value), interpolates the centroid weights compared to the fraction of the total weight defined by the input value to compute the estimate of the result quantile.



$k_\delta(q) = \delta((sin^{-1}(2q-1)+\pi)/2\pi)$. The size of each subsequence is smaller (i.e., more resolution) for centroids near the beginning or the end of the dataset, but larger towards the middle. This strategy tries to make queries for extreme quantiles, in general, more accurate than the ones closer to the median for outlier detection purposes. The construction process of t-digest, illustrated in Fig. 4.3(a), is closely related to the process of merging two sketches (Fig. 4.3(b)). The construction of one sketch requires merging a dataset (the elements correspond to centroids with weights equal to 1) against an empty t-digest. This process consists of sorting the weighted centroids and performing the grouping of subsequences as before but considering the given weights. To perform the query for a quantile we divide the weight of each centroid into two equal parts to the left and the right of the centroid. The quantile query receives the desired q and loops through the ordered list of centroids accumulating all the weights that have already been seen and comparing it to $q * |D|$, where $|D|$ represents the sum of weights (size of the dataset). If the desired weight ends up on a centroid, the value of that centroid is returned. This happens in the median query $qnt(0.5)$ in Fig. 4.3(c). On the other hand, if the weight ends up between two centroids, the value of the quantile is derived by linearly interpolating the values of the corresponding centroids using their weights (e.g., $qnt(0.6)$). The *cdf* query is implemented as the inverse of the result of a quantile query. Counting queries are also supported and return the sum of the weights of each centroid.

The publicly available implementations of t-digest were designed for applications in a data streaming scenario with low memory constraints. Their large memory overhead makes them not adequate to be used in datacube structures. We propose an optimized method called p-digest that reduces the memory footprint of the previous implementations and, therefore, suitable for our applications.

4.6 Quantile Data Structure

In this section, we describe the Quantile Data Structure (QDS). We present the queries supported, its internal representation, query algorithm, and implementation details.

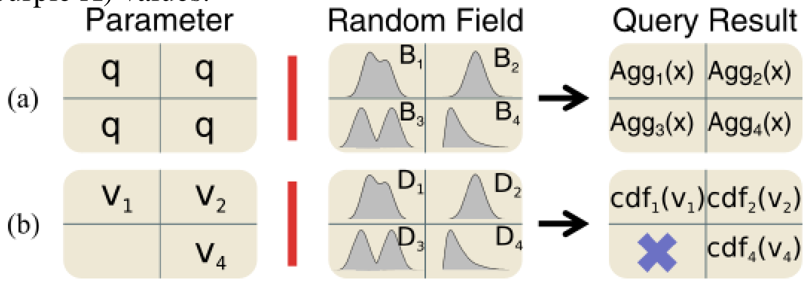
4.6.1 Overview and Query Types

Consider, for example, a hypothetical scenario of the analysis of flight delays in U.S. airports. We are interested in answering questions like T_1:*"How likely is a flight operated by Delta Airlines to be delayed more than 10 minutes at JFK airport?"*, T_2:*"How does the distribution of flight delays for two airports compare to each other in the past month?"* and T_3:*"How unusual are the delays experienced by Delta flights on January 29th, 2017?"*. To answer such queries QDS stores a quantile sketch as a payload at each node of a datacube to allow fast data selection and accurate querying for distribution statistics. QDS supports the following primary type of query:

select AGGR from QDS where CONSTRAINTS [group BY G]

The CONSTRAINTS part of the query represent conditions defined on any set of the *index dimensions* (e.g., carrier=Delta, airport=JFK) and specify the datacube nodes to consider. QDS groups these nodes into bins according to the group by dimension G (or create one group for all nodes if this optional information is not given). The quantile sketches associated with each datacube node in each group are merged to represent the distribution of the data in each group. In case a group by dimension G is specified, we merge sketches according to the bins of D forming a random field. For example, in our flight's scenario, grouping by the airport dimension will result in a collection of p-digests associated with each airport. Similarly, grouping by a temporal dimension with a given time granularity will result in a p-digest associated with each timestamp. The same is valid to spatial grouping (e.g., map tiles with resolution = 8 produces a maximum 256x256

Figure 4.4: Queries supported by QDS. Let F_M be the random field formed by merging quantile sketches of selected bins. (a) The quantile and cdf queries receive a parameter x and returns the result of the corresponding query for each quantile sketch. (b) The pipeline query: we use the result of a given query as a parameter to a second one using a right join process. In case the parameter has "missing" bins the result query can have undefined (purple X) values.



p-digest per tile). The aggregation function (AGGR) is executed on a *measure dimension* (e.g., arrival delay) and defines the quantile sketch query we execute on the random field: *quantile, cdf or count*. This process is illustrated in Fig. 4.4 (a).

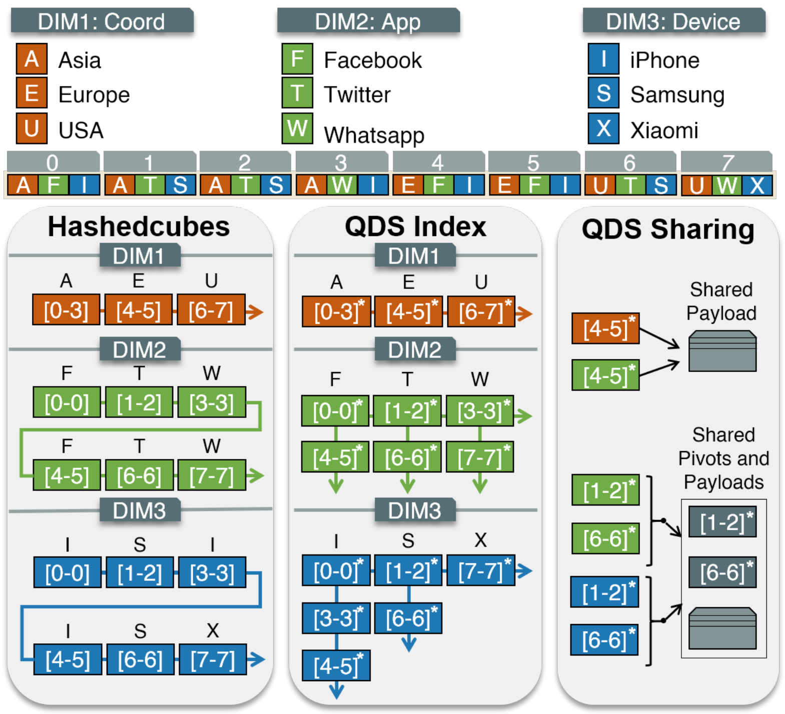
Quantile and *cdf* queries can answer questions T_1 and T_2 above. To answer question T_3, we use another query called *pipeline* (Fig. 4.4 (b)), which use the output of a query as input to a second one. In the T_3 example, we perform the first query to select all flight delays for Delta on January 29th. Let a second query select the total distribution of flight delays by Delta. QDS performs an operation of *right join* between the bins resulting from these two queries and compute the aggregation of the second query for each value in the output of the first query. The result is a score quantifying the *cdf* for January 29th in each airport. As described in Sec. 4.7.3, pipeline queries are the base for our event detection method.

The last query type supported by QDS is used to quantify the total deviation from the median over a period of time. Given a start/end timestamp and a temporal resolution (e.g., days) this query performs a set of pipeline queries for each timestep. In each timestep, the values are added up to create a score for each temporal bin. We name these *composite queries* and give examples in the use cases of Sec. 4.8. A detailed description of the execution of the query algorithm is presented in Sec. 4.6.3 and illustrated in Fig. 4.6.

4.6.2 Internal Representation

Datacube-inspired structures have as a common challenge the need to store data as compressed as possible while supporting fast query response. The exploration of data with

Figure 4.5: QDS indexing scheme and shared pointers. We use an example to compare the indexes of HC (a) and QDS (b). The input dataset has eight records, each with three dimensions. In HC, each dimension stores pivots in a pivot array that refers to intervals in the input dataset. In QDS, in addition to the pivot array for each dimension (primary pivot array), we keep a secondary pivot array for each element. In graphical terms, the primary array is displayed horizontally, while the secondary array is displayed vertically. Searching for values equal to F in QDS can be simply done by following vertically the secondary pivot array associated with F in dimension 2. The number of pivots stored in the QDS is not larger than in HC. Each pivot has an additional payload (marked with *) that can store quantile information. (c) Pivot arrays tend to have duplicate information across dimensions. To save memory, QDS used shared pointers to compact shared pivot and payload information.



primary index (a) primary/secondary index (b) shared data (c)

order statistics creates additional challenges. We describe below the indexing scheme, compression of shared information, and p-digest sketch that stores quantile data.

4.6.2.1 Indexing Scheme

The design of QDS is inspired by Hashedcubes (HC) because it offers the best trade-off regarding storage and efficiency. Since both structures have similar concepts, it is important first to review the design of Hashedcubes. To do so, we will use a simple dataset containing eight records (labeled from 0 to 7), each containing three categorical dimensions (location, app, and device) shown on the top of Fig. 4.5. Following a pre-defined ordering of the dimensions of the input dataset, HC keeps a multi-level index. For each dimension, this index stores an array of pivots that delimits a consecutive interval in the sorted input array with equal values. Fig. 4.5(a) shows an example of the index (pivot arrays) created with our sample dataset. In dimension 1 (location), one entry in the array has a pivot $[4 - 5]$ associated to the value E (Europe), meaning that in the input array, entries from 4 to 5 have values E in the first dimension. Observe that at dimension two there is more than one pivot associated with the values F (Facebook), T (Twitter) and W (Whatsapp). As a result, a query for F in the second dimension must find all its non-contiguous pivots. This is a simple example of *pivot fragmentation* which is a consequence of the multi-key sorting of pivot arrays in each dimension. As more dimensions are used, fragmentation increases, which causes queries that use subsequent dimensions to examine a possibly considerable number of pivots We experienced this corner case when implementing HC (see Sec. 4.9).

QDS's novel pivot index (Fig. 4.5(b)) fixes the fragmentation issue as well as supports the varied set of queries described previously. Starting at the second dimension, instead of a single pivot array, we keep an additional *secondary pivot array* that can be used to recover all pivots associated with a given value. For example, searching for values equal to F in the second dimension can be done by following the secondary pivot array associated with values F, which return the pivots [0-0] and [4-5]. The secondary pivot array allows keys associated with pivots to be stored only once, thus saving memory. Another improvement is related to the fact that pivot arrays for distinct dimensions in HC often have duplicated entries, leading to redundant storage. QDS overcomes this problem with a shared container abstraction. For example, in Fig. 4.5(b) we have the pivot [4-5] appearing in dimensions 1 and 2. Using a shared abstraction we create a single payload that is shared for both pivots, as show in Fig. 4.5(c). A second, and more sophisticated sharing happens when the secondary array is identical for different dimensions. In Fig. 4.5(b), the secondary array associated with the value T in the second dimension has pivots [1-2] and [6-6]. Similarly, the secondary array associated with the value S in the third dimension also has pivots [1-2] and [6-6]. In such cases, we share both the payload as well as the pivots, as shown in Fig. 4.5(c). We refer to Sec. 4.9 for experimental results of memory saved by these optimizations.

4.6.2.2 The p-digest data sketch

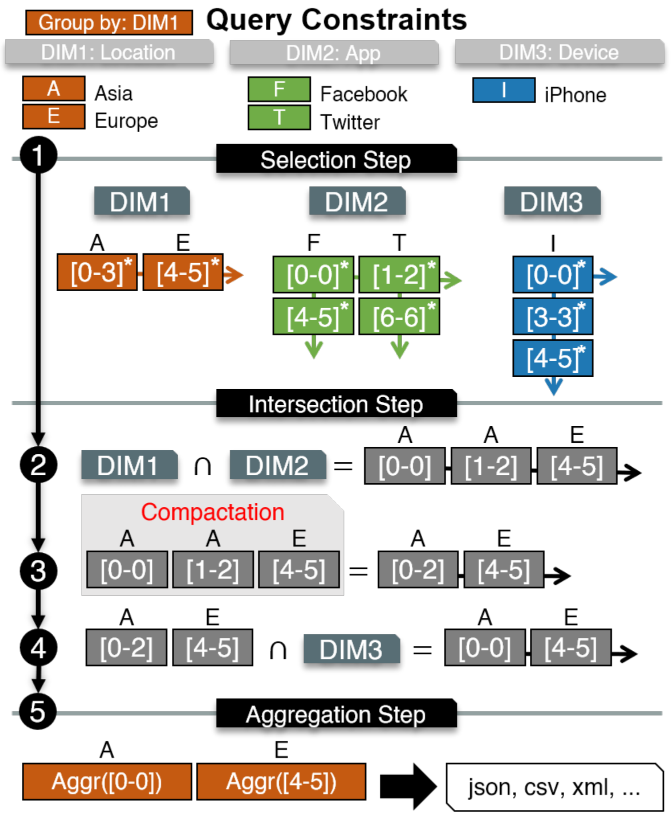
The t-digest described in Sec. 4.5 was our choice for storing payload information because it supports compressed and accurate on-line order statistics. There are, however, limitations in the two publicly available implementations of t-digest. The main implementation, described in (DUNNING; ERTL, 2014), uses a balanced binary search tree (AVL) to store centroid information, consuming 80 bytes per centroid. A secondary (and under construction) implementation uses an array, which reduces memory usage to 40 bytes per centroid. Such memory requirements are adequate for the streaming processing applications of t-digest, but in our datacube scenario, it results in prohibitive memory usage.

We made several changes to the array implementation of t-digest to comply with our performance requirements, For instance, we reduced the centroid memory storage by implementing the sketch as a stream of numbers, with both $centroid$ and $weight$ arrays as a single chunk of floats. The memory requirements for the centroid is at most 8 bytes, using 4 bytes for each of the centroid and weighted arrays. Using QDS with real data, a situation that frequently occurs is the weight array have all values equal to 1. To leverage this property and reduce memory usage, we added a boolean field to the end of the payload structure to indicate the storage of both centroid and weighted arrays. When this field is 0, the weight values are all equal to 1, and weights are not stored explicitly, only the centroid values. On average the cost for centroids is just 4 bytes. Similarly, we do not store the $weight$ array when all its values are equal. Such optimization is efficient for (very) small pivots in deeper dimensions. The $merge$ and $query$ operations were modified to work with this modified structure. For convenience, we call this modified structure by the name *p-digest*, since in QDS it associates one such sketch to each pivot. We implemented p-digest as a standalone library that can also be used outside QDS, which is available as an alternate implementation of t-digest (Sec. 4.6.4).

4.6.3 Query Algorithm

While QDS's and Hashedcubes's indices use similar concepts, their structural differences and the sophisticated set of queries supported by QDS makes querying our structure a very different process. QDS's query algorithm (Fig. 4.6) is responsible for efficiently selecting nodes and satisfying a set of query constraints. This algorithm was designed to handle a great variety of query combinations following a progressive refinement approach. In a high-level description, for a given multi-dimensional query, the query algorithm is composed of three steps executed in sequence: *selection, intersection, and aggregation*. In the *selection step*, for each dimension specified in the query, the algorithm selects the pivots from the pivot arrays that satisfies the query individually for each

Figure 4.6: The QDS query algorithm demonstrated using the dataset of Fig. 4.5. The input query has constraints in all dimensions. In the *selection* step, the pivot array of each dimension is processed to check the pivots that satisfy the query for that dimension. In the *intersection* step, we compute in sequence the intersection of the results of previous steps. The *aggregation* step compacts the results of the previous step.

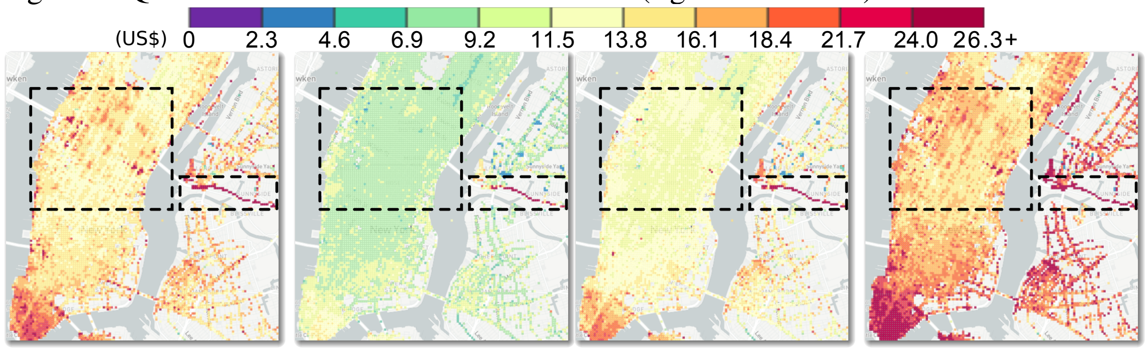


dimension. The primary and secondary pivot arrays are responsible for efficiently discarding queries that return empty results, thus avoiding the HC corner cases mentioned before (see a discussion on Sec. 4.9). The *intersection step* is responsible for combining the pivots, resulting from the selection step, that simultaneously satisfies the query for all dimensions. The result of the intersection step are pivots that might not be contiguous since the previous steps might leave similar elements distributed over distinct pivots. The *aggregation step* groups pivots by compacting disjoint pivots that contiguously represent the same value. For example, if pivots [1-2] and [3-5] refer to the same value F, we replace by a single pivot [1-5] of value F.

4.6.4 Implementation

QDS is implemented in C++ and uses a client-server architecture. The server consumes an input, and builds a QDS in a pre-processing step. QDS supports multiple categorical, temporal and spatial dimensions for its indexing schemas. We discretize spa-

Figure 4.7: Quantile heatmaps of taxi trip fares in NYC during the month of October 2014 based on their pick-up locations. The mean based heatmap (a) conveys high prices similar to the third quartile map(d). The median heatmap shows lower fares (c) indicating the robustness with respect to outliers. The first quartile map (b) indicates mostly lower values except on regions close to the Queens–Midtown Expressway near the high traffic region of Queens—Midtown Tunnel's toll station (right dashed box).



(a) mean (b) $q_1(0.25)$ (c) $q_2(0.5)$ (d) $q_3(0.75)$

tial and temporal dimensions like in NC or HC: quadtree based map tile coordinates and user-defined time bins, respectively. Unlike NC and HC, QDS can stack and intercalate different types of dimensions without a predefined order (e.g, categorical-temporal-spatial, or even, NC and HC ordering of spatial-categorical-temporal), since it impacts both memory usage and running time. Note that QDS default layout is the inverse of both NC and HC, since we find this to be a good compromise between performance and memory usage (refer to Sec. 4.9). We also support multiple measure dimensions by storing unique combinations (i.e., pivots) into individual primitive arrays (referenced as *payloads*). Each payload uses a header to determine the beginning and end of each dimension. Other low-level QDS optimizations are accessible in its open source code available at <https://github.com/cicerolp/qds>.

4.7 Building Visualizations with QDS

We illustrate below general scenarios of analytical tasks and visualizations enabled by QDS to support the visual analysis of large datasets with order statistics data.

4.7.1 Extending Usual Visualizations

The query capabilities of QDS support the analysis of data distribution patterns in spatial, temporal and categorical dimensions. For example, we define *quantile heatmaps* as heatmaps obtained from QDS's quantile queries. Fig. 4.7 compares the standard mean heatmap (a) against quartile queries (b,c,d) of taxi trip fares (in US dollars) in NYC based on their pick-up locations. We notice how the quantile heatmaps convey a different message than the mean heatmap. While the mean map (a) suggests high prices (above 16 dollars) for the region of Midtown (left dashed boxes), both maps of the first quartile (b) and median (c) suggest that these prices are usually smaller in that region (below 14 dollars). Also, notice how the mean map is similar to the third quartile map (d). This reflects the sensitivity of the mean to outliers. Furthermore, by performing a simple arithmetic operation on quantile heatmaps, we visualize spatial properties of the underlying distributions such as interquartile range (a robust alternative to variance as a measure of spread/uncertainty) and skewness (a measure of asymmetry in the distribution) (KENNEY; KEEPING, 1954). Another novel notion of heatmap enabled by QDS is called *cdf heatmaps*. These maps use the *cdf* query to display how likely a distribution in a given location is to be smaller than a certain value. Fig. 4.1 shows examples of this concept. The color on the maps represent the probability of flights being late, *i.e.*, $cdf(0)$. These maps make it intuitive to observe the changes in the geographical delay pattern from east to west in 2014.

The temporal aspect of quantiles can be explored for example by constructing *band plots* (Fig. 4.1 bottom). The median (black) curve gives a robust notion of centrality and therefore the typical temporal behavior of the variable in consideration. The curves of the first quartile (bottom curve) and third quartile (top curve) form dark red bands. The lighter red band is formed by quantiles 0.1 (bottom) and 0.9 (top). This choice was made to avoid minimum and maximum outlier values. Notice how the additional quantiles help in the identification of the variability of the distribution. Finally, we also notice different forms in the average and error bar based temporal plots (produced by GC (Wang et al., 2017)). The bands formed are not necessarily symmetric around the median curve and are a more faithful representation of the distribution behavior over time.

As the last example, QDS can be used to understand the distribution of quantitative values related to categorical dimensions. In fact, this can be done by using the quantile information to build the widely used box plots (Fig. 4.8) and *equi-depth histograms* (Fig. 4.2(d)). Unlike conventional histograms, the bins in equi-depth histograms contain a fixed fraction of the data population (equally spaced quantiles). In Fig. 4.2(d) the bins are colored proportional to their data density to better depict the data distribution.

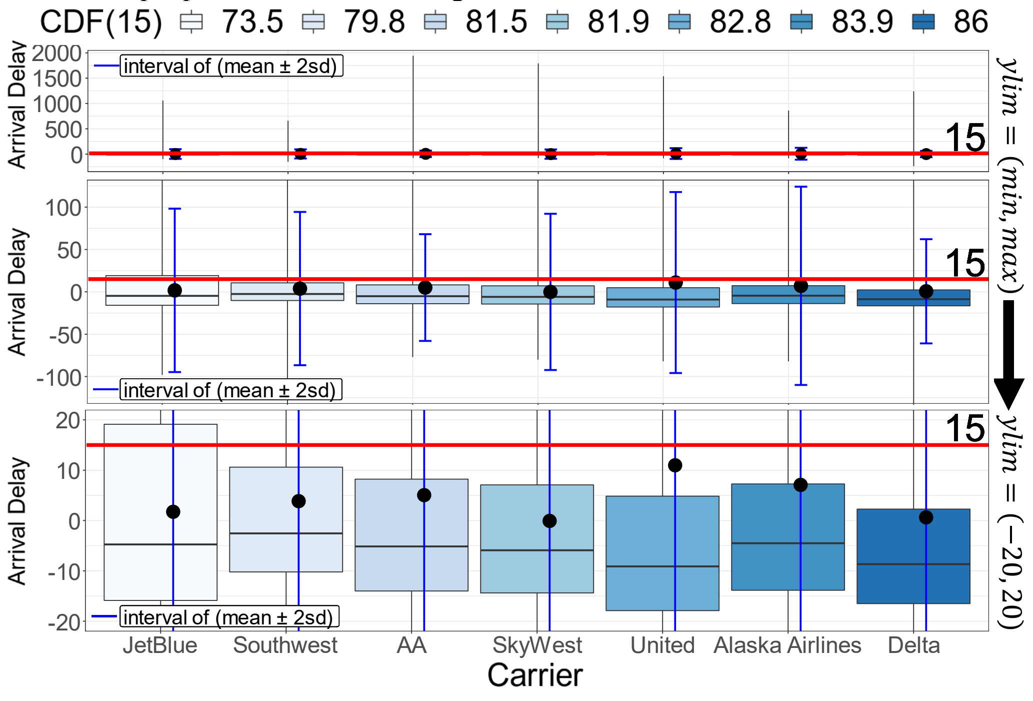
4.7.2 Easing the Reading of Uncertainty Visualizations

Interpreting uncertainty visualizations is not an easy task, even for trained individuals, One issued for this difficulty is performing statistical inferences by eye to quantify the uncertainty related to analytical tasks. An example of such scenario can be seen in Fig. 4.8. How likely is it for each of the distributions to be smaller than the red line (which represents the threshold of considering a flight to be late)? To simplify this problem, Ferreira at al. (FERREIRA; FISHER; KONIG, 2014) proposed interactive annotations that enrich usual uncertainty visualizations by visually quantifying uncertainty. One of these annotation allows the user to drag a line and the likelihood of the distribution to be smaller than this line would be mapped to colors. They provide a user evaluation that indicates the effectiveness of annotations in the sense that it improves "justified confidence", *i.e.*, the correlation between the user being correct and being confident her answer. However, Ferreira at al. (FERREIRA; FISHER; KONIG, 2014) did not propose an efficient data handling method to support these interactions. In fact, they used an ad-hoc sampling scheme which neither scales with the number of distributions nor supports slicing and dicing. Therefore it can not be used in a real visual analytics system. QDS can be used to support the interaction described above: it suffices to use the *cdf* query in each box plot with the value represented by the red line as parameter. The results of these operations are used to color the box plots in Fig. 4.8.

4.7.3 Uncovering the Unexpected

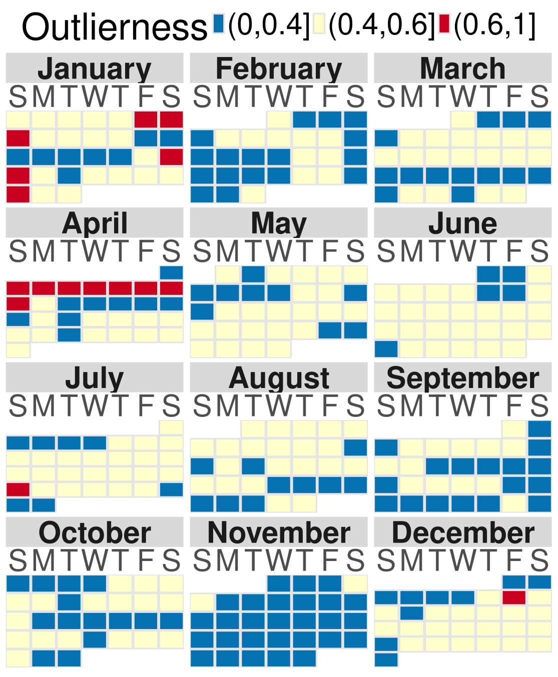
Performing visual exploration on large amounts of spatiotemporal data can be a time-consuming process. In fact, due to the inherent complexity of this data unusual (and possibly interesting) patterns might occur at multiple aggregation scales and therefore finding them requires users to inspect a large number of data slices over time and space. Thus, these patterns might remain undiscovered even after the use of visual exploration tools (DORAISWAMY et al., 2014). For this reason, the application of event detection techniques is essential to find these patterns. QDS's ability to retrieve the (approximate) distribution to an arbitrary portion of the data interactively is a powerful tool to perform event detection in a visual analytics system. In fact, given a value t and the distribution function F_X of a random variable X, we can define a measure of outlierness of t concerning F_X as (FRAIMAN; MUNIZ, 2001): $\hat{\phi}(t, F_X) = 2(|0.5 - F_X(t)|)$. A low value

Figure 4.8: Box plot of flight arrival delays per carrier. The boxes are colored and sorted according to the probability of each distribution being below 15 minutes (red line), which represents the proportion of on-time flights.

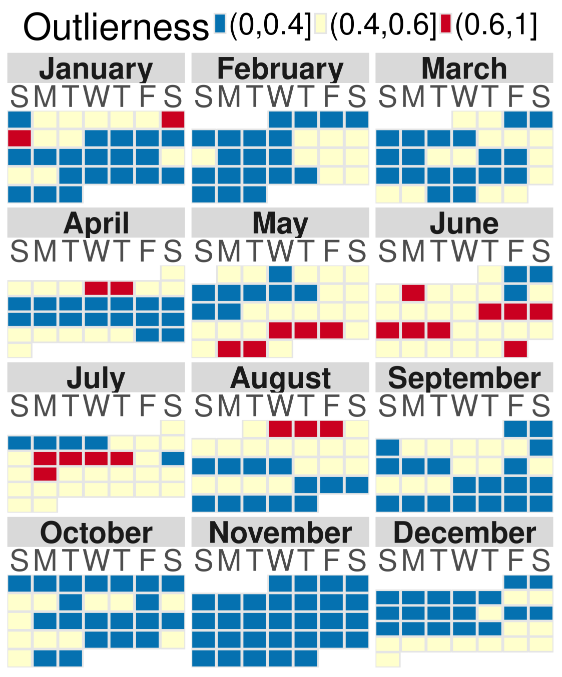


of $\hat{\phi}(t, F_X)$ means that t is a "normal" data instance, while high values mean instances closer to extreme values of the distribution and therefore judged as "events". Fraiman and Muniz (FRAIMAN; MUNIZ, 2001) proposed a method to extend this measure of outlierness to higher dimensional data. To describe this extension, we use as an example a heatmap m (analogous to t in the unidimensional case) of prices of taxi trips similar to the example given in Fig. 4.7(a). The function m assigns, for each geographical location p, a value $m(p)$, corresponding to the average price of taxi trips starting from that location. We assume to be given the random field F_M of the prices of taxi trips for every geographical location (analogous to F_X). We define the outlierness of m concerning F_M as $\phi(m, F_M) = \int \hat{\phi}(m(p), F_M(p))dp$, where the integral is taken over all points p on the map domain, and $m(p)$ and $F_M(p)$ denote the value of the heatmap m and the distribution of fare values at location p respectively. We compute the value of $\phi(m, F_M)$ using QDS's pipeline query described in Sec. 4.6.1. We choose the geographical coordinates as the group by dimension. In this manner, we can perform a *cdf* query for each $tile$ on the map. To obtain the final result, we simply compute $\hat{\phi}$ on each of these values and add up all the results. Such an approach can be used to find events in datasets with a long temporal range (Fig. 4.9).

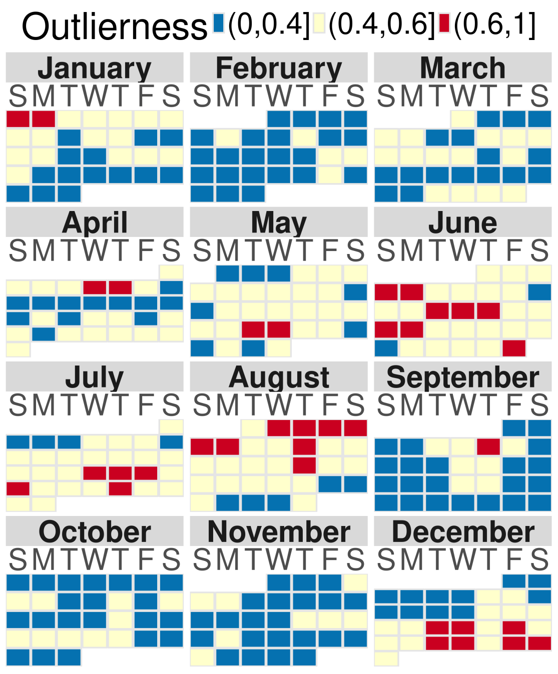
Figure 4.9: Daily arrival delay outlierness in 2017 for Delta JetBlue and Southwest airlines. Delta had an abnormal first week of April due to severe weather in its hub city Atlanta. Similarly, weather events created abnormal arrival times for JetBlue and Southwest in May and August respectively. January 29th is another odd day: a computer outage grounded all Delta's flights. The news on the side corroborate the unexpected events found.



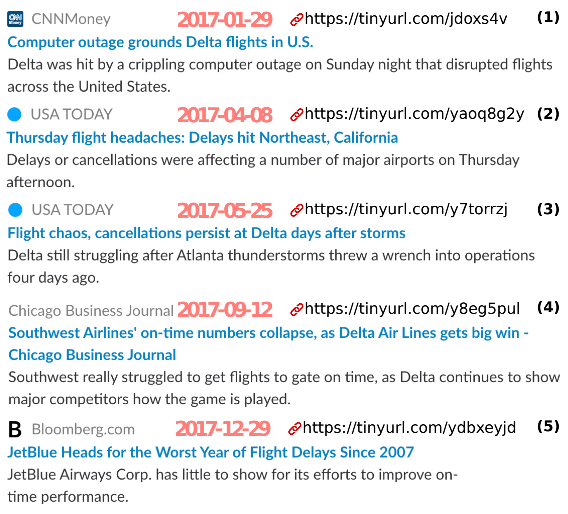
(a) Delta



(b) JetBlue



(c) SouthWest


CNNMoney 2017-01-29 https://tinyurl.com/jdoxs4v (1)
Computer outage grounds Delta flights in U.S.
Delta was hit by a crippling computer outage on Sunday night that disrupted flights across the United States.

USA TODAY 2017-04-08 https://tinyurl.com/yaoq8g2y (2)
Thursday flight headaches: Delays hit Northeast, California
Delays or cancellations were affecting a number of major airports on Thursday afternoon.

USA TODAY 2017-05-25 https://tinyurl.com/y7torrzj (3)
Flight chaos, cancellations persist at Delta days after storms
Delta still struggling after Atlanta thunderstorms threw a wrench into operations four days ago.

Chicago Business Journal 2017-09-12 https://tinyurl.com/y8eg5pul (4)
Southwest Airlines' on-time numbers collapse, as Delta Air Lines gets big win - Chicago Business Journal
Southwest really struggled to get flights to gate on time, as Delta continues to show major competitors how the game is played.

B Bloomberg.com 2017-12-29 https://tinyurl.com/ydbxeyjd (5)
JetBlue Heads for the Worst Year of Flight Delays Since 2007
JetBlue Airways Corp. has little to show for its efforts to improve on-time performance.

(d) Flight delays news

4.8 Use Cases

We demonstrate the capabilities of QDS in real exploration scenarios. We obtain all analyses while exploring datasets with millions of records interactively in a prototype visualization system using QDS, as can be seen in the demo video[1]. In all use cases we used p-digest's compression parameter $\delta = 50$ for the QDS construction.

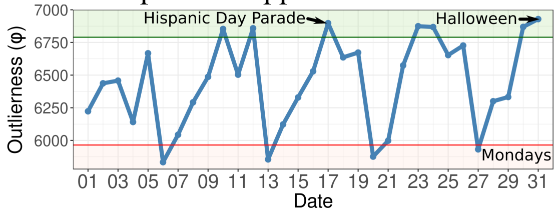
4.8.1 Analyzing Flights Delays

Delayed flights have a large impact on the finances of air carriers. According to the trade group *Airlines of America* each minute of delay costs around \$62.55 to U.S. airlines (Airlines for America,). Such costs represent a significant loss considering that some airlines accumulate millions of delay minutes every year. The *On-time Performance* dataset made available by the U.S. Department of Transportation (US Department of Transportation,) tracks the delays of U.S. air carriers domestic flights. This dataset has over 178 million flights in 30 years (1987 to 2017). To analyze this data we built a QDS structure on 9 of the 29 original columns of the dataset. As part of the index scheme, we used the categorical dimensions *canceled, diverted, carrier, departureAirport*, the temporal dimensions *departure time*, and *latitude and longitude* as the spatial dimension. We used the *departure delay* and *arrival delay* as payload dimensions.

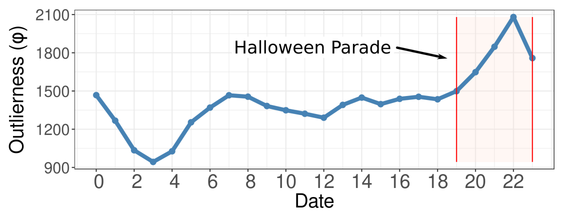
The *U.S. Bureau of Transportation Statistics* (USBTS) publishes periodic reports of carrier on-time performance. In these reports, a flight is on-time if it arrives no later than 15 minutes of its scheduled time. Fig. 4.8 shows a box plot of flight arrival delays distributions for some U.S. airlines, obtained using the QDS, using the data from the January 2017 through October of the same year. We use the sliding line interaction described in Sec. 4.7.2 to quantify the proportion of delayed flights according to the 15 minutes threshold. The results of $cdf(15)$ are used to color the boxes in the box plots. We also used them to sort the boxes in ascending order, to rank the airlines according to their delays. An interesting observation is a bad performance represented by the company JetBlue, for which 2017 was the worst year of flight delays in the previous decade (Fig. 4.9(d)-5). Also, we highlight that the ranking of carriers obtained by QDS matches the one reported by the USBTS for the period and the inferred densities are very close to the ones reported (US Bureau of Transportation Statistics,).

[1]<https://youtu.be/WSzTJXIVUw4>

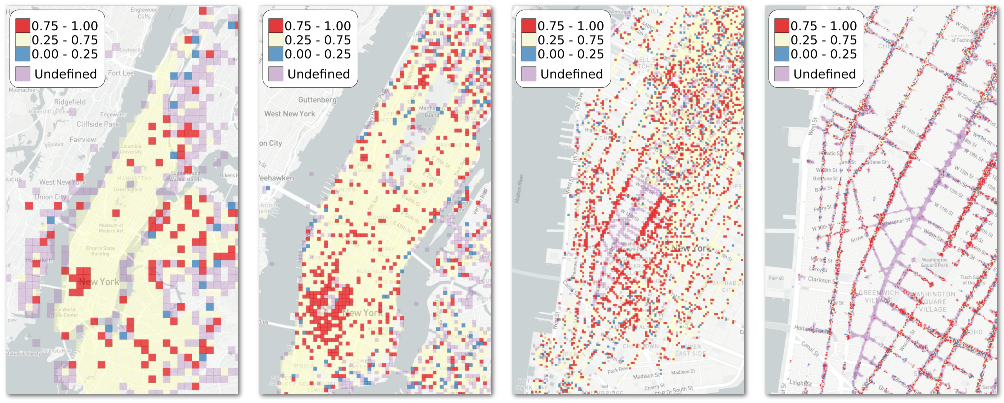
Figure 4.10: Exploration of NYC yellow taxi trips in October 2014. (a) Outlierness coefficients with respect to total fare vary widely during the month with peaks on days 10, 12, 17, 23, 24 and 31. This last one being the highest. (b) Analyzing how the outlierness vary over the day 31 we see that the day got more "unusual" with the highest values on the period starting at 7 PM. (c) The heatmap resulting from the pipeline query in this period we observe from left to right that trips are more expensive than normal in the Greenwich Village Region. Zooming in we see that a portion of the streets (purple) that unusually did not have any trips. This corresponds to the area where the annual Greenwich Village Halloween parade happened.



(a) Daily Outlierness in Oct. 2014



(b) Hourly Outlierness on Oct. 31 2014



(c) Heatmaps showing composite cdf query results for 31 Oct. 2014 - 7 to 10 PM

We study the JetBlue, Southwest, and Delta airlines to understand events that affect their delay patterns. For each company, we use QDS's outlierness query (Sec. 4.7.3) to quantify how unusual one day is if compared to the distribution of delays over the entire year. The results of this analysis are presented in Fig. 4.9. Days colored in red, yellow and blue have high, medium, and low measures of outlierness respectively. An interesting case is the entire red week for Delta at the beginning of April. This odd week for the company was caused by severe thunderstorms that happen in Delta's hub city of Atlanta. During this week more than 3000 Delta flights were canceled (Fig. 4.9(d)-2). At the end of May, JetBlue had delays due to heavy rains in the Northeast of the U.S, leading to cancellations in main airports for JetBlue in New York City and Boston (Fig. 4.9(d)-3). Southwest experienced unusual delays in August due to Summer thunderstorms, the Hurricane Harvey, and the high seasonal demand (Fig. 4.9(d)-4). While weather is the main cause of flight delays in the U.S., we found an equipment malfunction event (a computer outage) that grounded all Delta's domestic flights on January 29 (Fig. 4.9(d)-1).

Table 4.1: Overall summary of the relevant information for building QDS.

dataset	size	index schema(bits)	payload schema	QDS Memory/Time		HC Memory/Time
				leaf-size $= 1$	leaf-size $= 32$ or 64	leaf-size $= 32$ or 64
brightkite	4.5 M	dayOfWeek (3), hourOfDay (5), time (16), lat (25), lon (25)	NA	455 MB/9s	276 MB/7s	366 MB/7s
gowalla	6.4 M	dayOfWeek (3), hourOfDay (5), time (16), lat (25), lon (25)	NA	711 MB/13s	367 MB/11s	743 MB/13s
twitter-small	210.6 M	device (3), time (16), lat (17), lon (17)	NA	3.1 GB/05:55m	2.7 GB/05:54m	4.9 GB/10:53m
twitter	210.6 M	app (2), device (3), language (5), time (16), lat (17), lon (17)	NA	4.6 GB/06:39m	4.2 GB/06:37m	9.4 GB/12:04m
flights	121.2 M	dep. delay (4), carrier (11), dep. time (16), lat (25), lon (25)	arrDelay, depDelay	1.4 GB/02:50m	1.4GB/02:50m	457 MB/03:56m
green-taxis-small	42 M	pickupDateTime (16), lat (22), lon (22)	ttlAmount, distance	1.3 GB/01:24m	1.2 GB/01:16m	788 MB/03:56m
green-taxis	42 M	dayOfWeek (3), hourOfDay (5), pickupTime (16), lat (22), lon (22)	ttlAmount, distance	1.3 GB/01:16m	1.2 GB/01:15m	3.0 GB/01:49m
yellow-taxis-small	706 M	pickupDateTime (16), lat (22), lon (22)	ttlAmount, distance	9.7 GB/27:53m	9.3 GB/28:04m	7.0 GB/18:14m
yellow-taxis	706 M	dayOfWeek (3), hourOfDay (5), pickupTime (day), lat (22), lon (22)	ttlAmount, distance	9.7 GB/31:37m	9.3 GB/31:33m	12.6 GB/20:38m

4.8.2 Exploring Outlierness in Taxi Trip Records

New York City (NYC) is one of the largest cities in the world. Its taxi system is a big part of the city's life, with more than 13 thousand cabs driving every day. The NYC Taxi and Limousine Commission have collected and distributed monthly yellow taxi trips records since 2009 (NYC Taxi and Limousine Commission,). We use QDS to analyze some of the fields in this dataset. The QDS index has pickup location (latitude and longitude) as the spatial dimension, pickup date-time as the temporal dimension and passenger count and payment type as the categorical dimension. As payload dimensions, we use the total fare and trip distance.

We describe interesting events that happened during October of 2014. For each day in that month, we computed the outlierness of the total fare of trips compared to the distribution of the entire month (Fig. 4.10). We observe that Mondays in that month (days 6, 13, 20 and 27) have the lowest outlierness (shaded red region). On the other hand, days colored in shaded green are on top of the outlier list, corresponding to Thursdays (23, 30), Sundays (12) and Fridays (10, 17, 24, 31). The top of the list a Friday (31). To justify this, we notice, for example, that Sunday 12 was the day preceding the Columbus day holiday on which some events changed the traffic on major avenues most of the day. In fact the CBGB Music Festival blocked a portion of Broadway from 10:30 AM to 19:30 PM and the Hispanic Columbus Day Parade closed a long portion of Fifth Avenue from noon to 5 PM (NYPD,). We explore the top candidate according to our outlierness metric, October 31. To investigate what makes this day stand out, we performed another outlierness time analysis now comparing each hour on this day against the distribution of total fare of all hours in the month. The resulting time-series can be seen in Fig. 4.10(b). We see that the outlierness attains its highest values at the end of the day starting at 7 PM. To understand what makes this hour to stand out, we use a composite cdf query to map how the distribution of these hours (7 PM to midnight) compares to the rest of the month. Colors in the map reflect the results of the composite cdf query: blue, yellow, red and purple mean low, medium, high and missing quantile values respectively. Looking at the map of the city (Fig. 4.10(c)) we see a large red region (trips more expensive than normal) on the Greenwich Village. Such trips have fares 75% more expensive than fares of the entire month. A zoom in this area shows progressively more details of this pattern, revealing that expensive trips happened around an area where no trips happened (purple region) during the interval from 7 PM to midnight. This area corresponds to a portion of the 6^{th} avenue, where the annual Greenwich Village Halloween parade happened in 2014.

4.9 Experimental Results

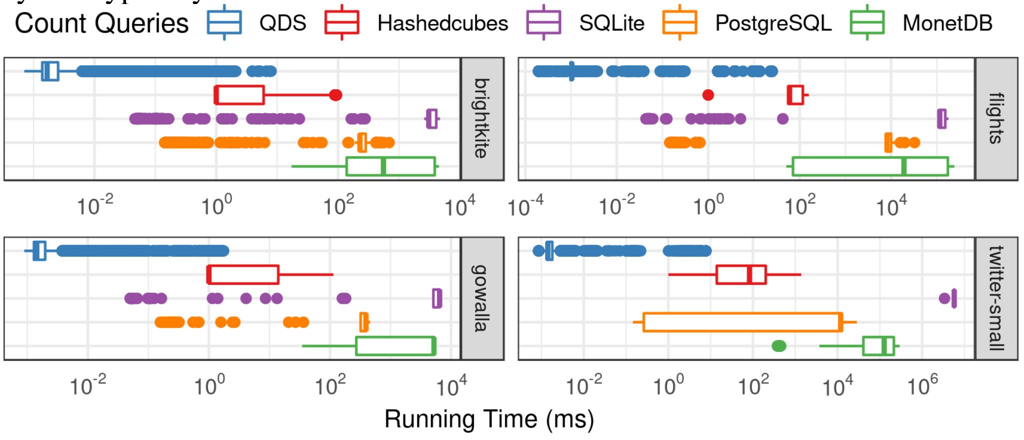
We evaluate our method in two sets of experiments. The first one evaluates the QDS index. We begin by performing a direct comparison to HC regarding construction time and memory usage for several datasets and schemas. Later we compare the response time of count queries in QDS as well as to the three commonly used databases SQLite, PostgreSQL and MonetDB. The second set of experiments evaluates the p-digest payload performance concerning approximation accuracy, memory usage, and computational performance. We compare against the quantile query capabilities present in the database solutions previously mentioned. The experiments were performed in a Linux-based machine, with an Intel Core i7 4790 with 32GB of main memory. We used the default options of the databases and the SQLite in-memory configuration. Benchmark data and code are available at the QDS's code repository.

4.9.1 The QDS Index Experiments

Memory usage. We compare the memory usage and construction time of QDS and HC for different datasets and schemas (Table 4.2). HC adopts a *minimum leaf-size* in its spatial dimension to improve query time and memory footprint. On the other hand, it leads to poor visual accuracy for regions with a low number of elements and, more importantly, for outliers. To enable a direct comparison with HC, for this benchmark, we build an experimental version of QDS with a modified implementation of both construction and query algorithms that integrate the minimum leaf-size technique. We notice that QDS's memory usage is comparable (and in some cases much better than) with Hashedcubes (*leaf-size = 32 or 64*). Regarding construction time, our method achieved better results than HC in most of the datasets considered even if we do not modify the leaf-size (*leaf-size = 1*).

Query Latency. We assess the performance of QDS's index by measuring its latency on

Figure 4.11: Performance comparison of QDS, HC and database alternatives computing count queries. QDS novel index successfully avoid HC corner cases and offers a query latency that typically lies below 10ms in various datasets.



the same 87449 spatio-temporal count queries (with spatial, categorical and temporal constraints) used on the HC paper. These queries were collected on the public NC site while users explored the *brightkite*, *gowalla*, *flights* and *twitter* datasets. We compare the results from QDS, HC and spatial extensions of SQLite, PostgreSQL and MonetDB (MonetDB B.V.,), using their recommend approach to accelerate spatial queries. To enable the direct comparison of data cube structures and database solutions, we translated the set of queries mentioned above to the appropriate format of each system. As observed in Fig. 4.11, QDS outperforms all the tested solutions with query latency typically lying below 10ms. We highlight that it successfully avoids HC corner cases.We notice that SQLite, PostgreSQL and MonetDB spatial indexes implementations were unable to offer efficient mechanisms to perform spatial filtering while combining categorical and temporal constraints. We specially notice that MonetDB does not support any special accelerators for spatial objects[2] and hence its poor performance. Overall, QDS's index offers real-time ($< 40ms$) slicing and dicing with ease, which we use to provide complex quantile queries at interactive rates (e.g., pipeline queries).

4.9.2 The p-digest Sketch Experiments

We now report accuracy, performance and memory usage of p-digest through five experiments. The first three of them evaluate the QDS's memory/accuracy trade-off introduced by p-digest's compression parameter δ. To evaluate this trade-off, we measure the quality of quantile estimation in different conditions of data compression, merge effective-

[2]https://www.monetdb.org/Documentation/Extensions/GIS.

ness and queried quantile, while varying δ. For this experiments, we used the *green-taxis* dataset to stress p-digest worst-case scenarios. The values shown in the accuracy experiments are computed by measuring the relative error of estimated quantile to the actual empirical quantile for the input data: $|q_{estimated} - q_{empirical}|/(|q_{empirical}|{+}1)$.

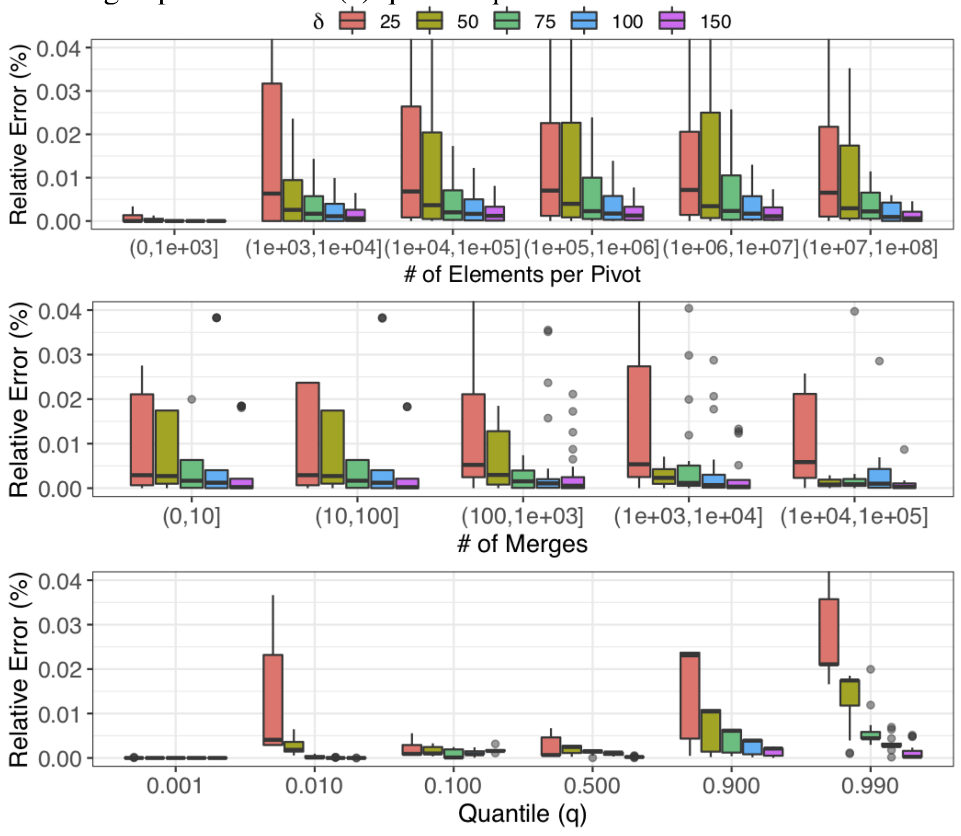
Accuracy per Spatial Quadtree Node. This experiment gives an insight into the accuracy of p-digest when dealing with input data that range from 1 element to 100 million elements. After loading the dataset into QDS, we query each region of the spatial index independently from `root` to `leafs` at height 25, measuring the accuracy during the process. This benchmark exploits p-digest capacity to approximate different set sizes. Fig. 4.12 (a) shows that quantile estimation is accurate for small input data and nearly unaffected for variations on compression parameter δ. The average error is somewhat constant for larger inputs.

Accuracy per Spatial Quadtree Level. In this experiment, we combine each quadtree level into its respective p-digest, i.e., for every level, we execute (at most) 4^Z p-digest merge operations while keeping the same input data. Fig. 4.12 (b) shows that accuracy increases when the input data is broken into more parts, because data is spread across more p-digest arrays. QDS benefits from this since pivot intersections (and as a result, p-digest merges) are commonly executed to answer the range of queries we support. Compression parameter $\delta = 50$ was the default value because it has the right balance between (i) accuracy per number of elements, (ii) accuracy per number of merges and (iii) memory usage.

Accuracy by Varying $q^{th} \in (0, 1)$. To measure accuracy on extreme quantiles, we merge each quadtree level into its respective p-digest and aggregate the estimated quantiles per q^{th} (Fig. 4.12 (c)). This experiment gives an insight into the error of a typical real-world query and the importance of the choice of the compression parameter δ. The relative error of compression parameter $\delta = 25$ gets worse near $q = 1$. This behavior reflects a poor choice of this parameter for the distributions that we are trying to approximate. Notice how the performance improves for larger values of δ.

Performance. We now compare the latency of QDS quantile queries against similar queries provided by SQLite, PostgreSQL and MonetDB. As a baseline for evaluation of p-digest, we also implemented an experimental variaion QDS, here referenced as QDS (w/o p-digest), that performs exact quantile computation. To do so we use QDS index and store at each pivot a sorted array containing the corresponding payload t values. This baseline give us an insight about the performance of QDS index when using a naive ap-

Figure 4.12: Evaluation of p-digest's quantile estimation with respect to (a) pivot size, (b) number of merge operations and (c) queried quantile.

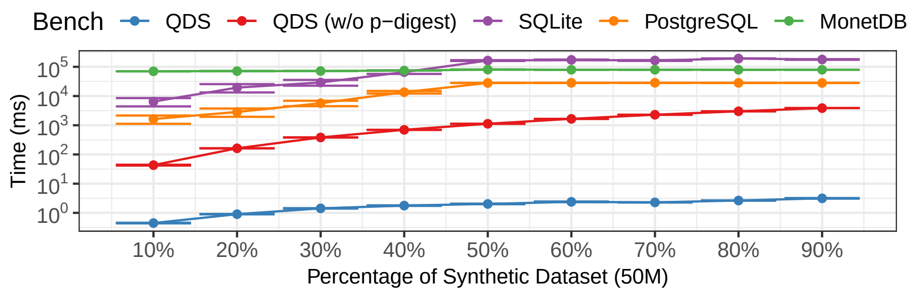


proach to calculate quantiles.

For this experiment we use a synthetic dataset composed of 50 million points (x, y, t), where the spatial dimensions x, y are independent and uniformly distributed in the interval $[0, 10]$. The payload dimension t is generated by sampling the standard normal distribution. The goal is to compute the median of payload values over the points contained in randomly generated spatial regions of varying size that covers from 10% up to 90% of the dataset domain. As shown in Fig. 4.13, SQLite and PostgreSQL employ different acceleration strategies but are unable to provide queries at interactive rates. As already stated, MonetDB lacks any sort of index to accelerate spatial queries, translating to a constant latency, no matter the number of filtered points. We highlight that all database solutions perform exact quantile computations, which makes necessary to scan all elements filtered by the spatial query. The regular build of QDS (with p-digest) was the only method able to provide quantile queries at interactive rates. QDS qnt computation time is dominated by merge operations. When measured individually, merging times were less than one millisecond while using compression parameter $\delta = 50$.

Memory usage. Memory usage was a relevant factor when designing p-digest. The

Figure 4.13: Comparison of $qnt(0.5)$ computation on a synthetic dataset using QDS, QDS without p-digest and database alternatives. As shown, QDS can provide quantile queries at interactive rates.



ability to share pivots and payload data, as well as memory saving strategies, prevent QDS size to be directly proportional to the p-digest *compression (δ)* parameter. While the average number of data items per p-digest is small, it is necessary to use a quantile sketch algorithm to enable quantile computation in large datasets, as shown in Fig. 4.13. As datasets become larger, allocating a buffer to store temporary data from the naïve approach becomes worse, since its size is proportional to the number of elements to compute the quantile. As observed in Table 4.2, the variation of *compression (δ)* parameter values has little impact on memory usage. Compression ratio ranges from $1.16x$ up to $1.34x$ when compared with the naïve solution.

4.10 Discussion

In this section, we further discuss issues related to QDS performance, applicability, and limitations. We first highlight that QDS improves on NC, Immens and Hashedcubes with respect to both capabilities, since these systems only support count queries and performance as described in Sec. 4.9. We also notice that having an (approximate) description of the distribution of a dataset is more powerful than using a parametric distribution, such as the Gaussians (used by GC). In fact, using p-digest we can retrieve the (approximate) values for moment statistics. However, the quantiles of a parametric distribution fitted to a dataset are in general far from the original ones (as illustrated in Fig. 4.8). This makes QDS widely applicable for the analysis of large spatiotemporal datasets. An interesting application scenario is the analysis of ensemble datasets in which different model predictions are put together to represent the diversity of the phenomenon under study.

A limitation of QDS compared to GC is the fact that it can only deal with univariate

Table 4.2: Compression results for different p-digest configurations.

dataset	p-digest compression							
	naïve		$\delta = 25$		$\delta = 50$		$\delta = 100$	
	memory	time	memory (compression)	time	memory (compression)	time	memory (compression)	time
flights	12.9 GB	05:35 m	9.9 GB (1.31 ×)	07:57 m	10.5 GB (1.22 ×)	08:09 m	10.9 GB (1.18 ×)	08:19 m
green-taxis	9.0 GB	01:54 m	6.7 GB (1.34 ×)	03:02 m	7.0 GB (1.30 ×)	03:05 m	7.2 GB (1.25 ×)	03:08 m
green-taxis-small	8.4 GB	01:52 m	6.7 GB (1.25x)	02:50 m	7.0 GB (1.20 ×)	02:54 m	7.2 GB (1.16 ×)	02:57 m
yellow-taxis-small	NA		12.7 GB (-)	54:57 m	12.9 GB (-)	55:59 m	13.3 GB (-)	56:04 m

distributions (due to a limitation of p-digest) and, therefore, treats its payload dimensions as independent variables. Finally, while we have shown that QDS achieves a good approximation, it does not provide error bounds. We intend to investigate how to quantify and communicate the uncertainty in the approximation to the user. Also, we want to perform a formal user study to evaluate the use of QDS and the supported visualizations.

4.11 Conclusions and Future Work

In this paper, we presented QDS, a fast and memory efficient data structure that supports real-time (virtually immediate feedback) data exploration based on order statistics on large multidimensional datasets. We believe that these capabilities open a large number of opportunities to design novel visual encodings and interaction techniques. In fact, other visualizations (matrix heatmaps, attributed network and etc) based on averages could be adapted to use their robust counterparts. Furthermore, we want to explore the possible use of QDS to speed-up computations in machine learning techniques for non-Gaussian distributions such as quantile regression and quantile based clustering. We see the coupling of cutting edge data sketching techniques with powerful precomputed indices to support interactive visual analytics as a promising future research direction. For example, we would like to explore how we can use of matrix and tensor sketching techniques to support the execution of complex analytical algorithms interactively. Another research direction is to define a data sketch to represent multivariate distributions with features similar to how t-digest can represent univariate distributions. To the best of our knowledge we are not aware of a solution to this problem. We believe the queries provided by QDS provide changes in the mindset during the analysis, allowing users to reason on the likelyhood of a hypothetical scenario like in Fig. 4.1. We intend to investigate these ideas in the future.

5 VISUAL FORMATION AND COMPARISON OF PATIENT COHORTS

Published in: The Proceedings of the VLDB Endowment (PVLDB), 2019.
Qualis CAPES: A1
Authors: Cícero A. L. Pahins, Behrooz Omidvar-Tehrani, Sihem Amer-Yahia, Valérie Siroux, Jean-Louis Pepin, Jean-Christian Borel and João L. D. Comba.

5.1 Abstract

We demonstrate COVIZ, an interactive system to visually form and explore patient cohorts. COVIZ seamlessly integrates visual cohort formation and exploration, making it a single destination for hypothesis generation. COVIZ is easy to use by medical experts and offers many features: (1) It provides the ability to isolate patient demographics (e.g., their age group and location), health markers (e.g., their body mass index), and treatments (e.g., Ventilation for respiratory problems), and hence facilitates cohort formation; (2) It summarizes the evolution of treatments of a cohort into health trajectories, and lets medical experts explore those trajectories; (3) It guides them in examining different facets of a cohort and generating hypotheses for future analysis; (4) Finally, it provides the ability to compare the statistics and health trajectories of multiple cohorts at once. COVIZ relies on QDS, a novel data structure that encodes and indexes various data distributions to enable their efficient retrieval. Additionally, COVIZ visualizes air quality data in the regions where patients live to help with data interpretations. We demonstrate two key scenarios. In the *ecological scenario*, we show how COVIZ can be used to explore patient data to generate hypotheses on the health evolution of cohorts. In the *case cross-over scenario*, we show how COVIZ can be used to generate hypotheses on cohort health and pollution data. A video demonstration of COVIZ is accessible via **<http://bit.ly/coviz-video>**.

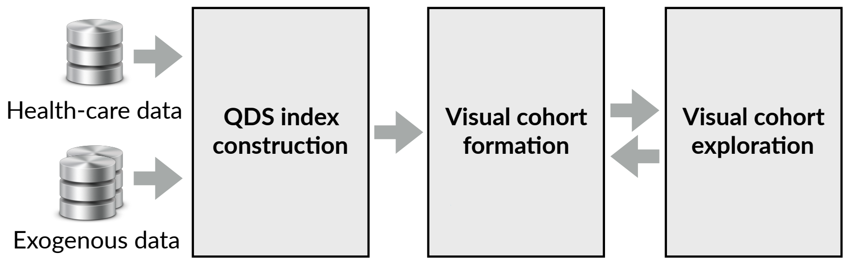
5.2 Introduction

With the increasing availability of large-scale health-care data in various sectors (e.g., prognoses, treatments, hospitalizations and compliances), medical experts need effective data-driven methods to identify patient cohorts, examine and explain their health and its evolution, and compare cohorts. Medical cohort analysis exhibits the collective behavior of patients, providing insights on the evolution of their health conditions and their reaction to treatments and to their environment (OMIDVAR-TEHRANI; AMER-YAHIA; LAKSHMANAN, 2018). Cohort analysis serves various goals such as augmenting treatment effectiveness, defining health campaigns and public policies, understanding patient satisfaction, and optimizing health-care spending and revenue (MUNSHI; SHARMA; SHARMA, 2017). The many facets that affect patients' health *require to adopt an exploratory and holistic approach to its analysis*. Medical experts *do not necessarily know what to look for* in the data, which cohorts are most insightful, and how to make sense of some observations. Cohort analysis can greatly benefit from *a visual tool that helps them walk through their data to identify cohorts of interest and generate hypotheses*. An essential aspect of that process is the ability to enrich observations with exogenous data that can be used to make sense of some phenomenon. For instance, analyzing data about patients suffering from respiratory problems would benefit from visualizing air quality data in the regions where those patients live.

We propose to demonstrate COVIZ, a system that acts as *a visual enabler* for cohort formation and exploration. COVIZ lets medical experts form cohorts, obtain their various statistics, examine their health condition and treatments, visualize how their health evolves over time, and compare cohorts. To do that, COVIZ relies on two principles: **aggregated analytics** and **interactivity**. Aggregated analytics refers to forming groups of patients (aka cohorts) in an exploratory fashion and observe their collective behavior. Cohorts can be formed with common demographics, health markers, and treatments. The visual interface helps medical experts examine different possibilities of forming cohorts, verifying members of cohorts, and examining differences in their health status. Interactivity requires fast iterations so that the train of thought of the analyst is not lost during the formation and exploration of cohorts. To ensure that, COVIZ relies on QDS (de Lara Pahins; Ferreira; Comba, 2019), a novel data cube structure that encodes various distributions of health-care data and indexes them to enable their efficient retrieval. To the best of our knowledge, COVIZ is the first mixed-initiative visual analytics system that enables medical experts to form and explore cohorts.

Visual analytics has been recently applied to enrich different data analysis tasks. Zenvisage (SIDDIQUI et al., 2016) enables visual querying of data, where experts need to express their needs in a SQL-like language which operates on top of a visual algebra to show results. Vexus (AMER-YAHIA et al., 2018) provides native support for visualizing and exploring groups of users. Vizdom (CROTTY et al., 2015) enables an interactive

Figure 5.1: COVIZ architecture.



whiteboard to compose complex workflows of data analysis and statistics. It exploits approximation and partial refinement techniques to deliver visualizations interactively. Also SeeDB (VARTAK et al., 2015) and Voyager (WONGSUPHASAWAT et al., 2016) are visualization recommendation tools that explore the space of visualizations, and recommend interesting ones. While interactivity has been the focus of these systems, there has been less attention towards aggregated analytics (i.e., analysis of cohorts). COVIZ is a single destination system that visually enables the formation and exploration of medical cohorts without the burden of formalizing queries. As such, it can easily be used by medical experts to identify cohorts of interest and generate hypotheses on their health evolution and the impact of the environment.

5.3 System Design

The overall architecture of COVIZ is illustrated in Figure 5.1. Initially an index is built offline to boost online cohort formation and exploration. COVIZ displays health-care data and other exogenous data sources as separate layers over a geographical map. A set of filters is provided in the visual interface to facilitate the visual formation of cohorts. Once a cohort is formed, COVIZ provides a succinct representation of the cohort's health trajectory which helps analysts comprehend the health evolution of cohort's members and compare cohorts (i.e., cohort exploration). COVIZ is a web service whose front-end is implemented in the Angular framework and back-end in C++ (the index) and Python (cohort exploration). The implementation of COVIZ is publicly available under GPL-3.0 license: **<http://bit.ly/coviz-code>**.

5.3.1 Datasets

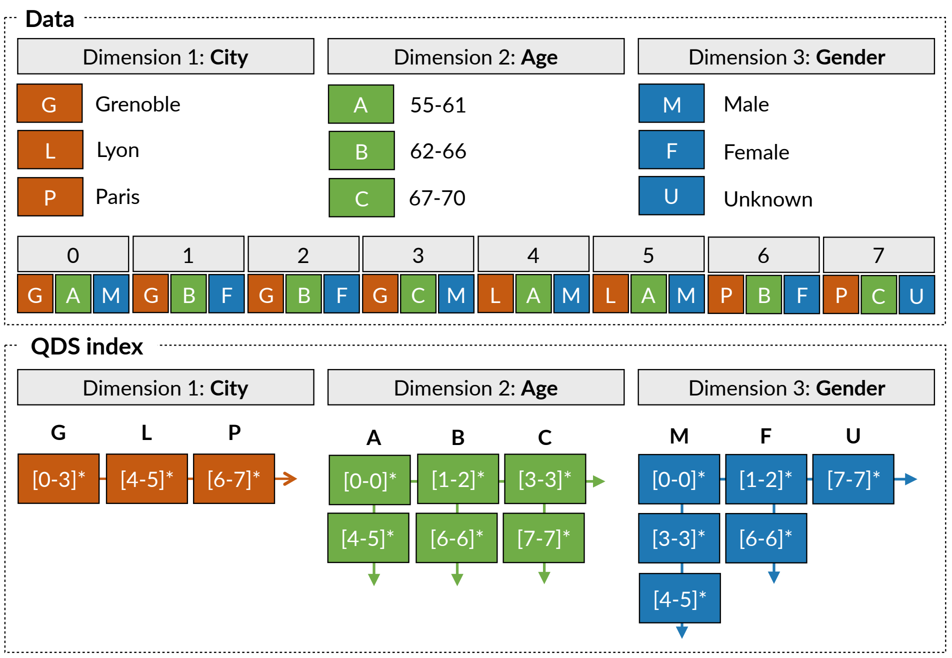
Health-care data. We use a dataset from our medical partner which contains events of $56,284$ patients with respiratory problems between the years 2000 and 2017. Patient events are: treatment, compliance, etiology, fatigue marker, BMI marker, sleepiness marker, and hospitalization. The dataset has $1,536,516$ records in the following schema $\langle patient\_id, lat, lon, date, marker, value, treatment\_\ durations\rangle$. Each record reports the *value* of a *marker* (fatigue, BMI, and sleepiness) for a specific patient identified by *patient_id*. Also $treatment\_durations$ reports the duration of treatments (with a month-level precision) which co-occurred with the *marker* for the patient *patient_id*. Examples of treatments are Aerosoltherapy (AERO) and Oxygenotherapy (OXY). Each patient is also associated with a set of demographics such as *gender*, *age*, and *life status*. Figure 5.3-A illustrates a visualization of health-care data where colors are mapped to the number of patients.

Pollution data. High air pollution levels can cause serious respiratory problems. We consider a dataset of air pollution as an exogenous resource to enable potential explanations of observations in the health status of the formed cohorts. The dataset contains values of different air pollutants (NO2, Ozone, PM2.5, and PM10) for all of France in the period of 2009 to 2013. The exposure models, developed in the context of a European project EU-FP7 SYSCLAD (AL, 2016), have a fine spatial resolution ($1km \times 1km$) and temporal resolution (on a daily basis). The dataset has 2,671,128,000 records in the following schema $\langle lat, lon, date, pollutant, value\rangle$.

5.3.2 Cohort formation

A cohort denotes a set of patients with common predicates (i.e., demographics, health markers, and treatments). For instance in our data, the cohort of female patients in Grenoble contains $1,531$ members whose predicates are defined on "gender" and "city" dimensions. To form cohorts, experts should be able to add/remove filters on predicates and the visual interface should provide immediate insights on the changes. The final set of filters will constitute the cohort. Cohort formation is not a straight-forward task for medical experts as they often have a partial understanding of their data and their needs. Hence they need to iterate over several exploration steps to reach their cohort of interest. This requires *interactive performance* to ensure a latency under $100ms$ (FEKETE; PRIMET, 2016). To achieve that, different indexing schemes have been proposed, all of which pre-compute statistics for some pre-defined aggregations, such as count and average (GANI et al., 2016). However most indexes store simple aggregations over individual data records. Hence they do not provide native support for cohorts and their detailed

Figure 5.2: An instance of QDS indexing scheme for eight records and three dimensions. QDS stores at each pivot (marked with an asterisk) a payload that contains the representation of a distribution function.



statistics. Moreover, the index structure should be adapted to the spatial aggregation of records.

COVIZ benefits from a new generation of data cube structures designed to support visual and interactive cohort formation by supporting *count queries* used in heatmaps and histograms, e.g., "how many events occurred in a given region on a given date?" (LIU; JIANG; HEER, 2013; Lins; Klosowski; Scheidegger, 2013; Pahins et al., 2017; Wang et al., 2017). COVIZ integrates a new data cube structure called Quantile Data Structure (QDS) (de Lara Pahins; Ferreira; Comba, 2019) which is an extension of Hashedcubes (Pahins et al., 2017) to improve efficiency and support a variety of aggregation queries, such as variance and quantile aggregations. Unlike count queries, such aggregation queries incorporate the inherent data distribution to provide more flexibility for cohort formation. At a high-level, QDS stores multi-dimensional data (spatial, temporal, and categorical) in an array ordered by a nested sorting in each dimension. The ordering allows the construction of a multi-level index that keeps, for each dimension, a list of intervals (called pivots) that delimit a consecutive region in the array. An illustration of QDS is provided in Figure 5.2.

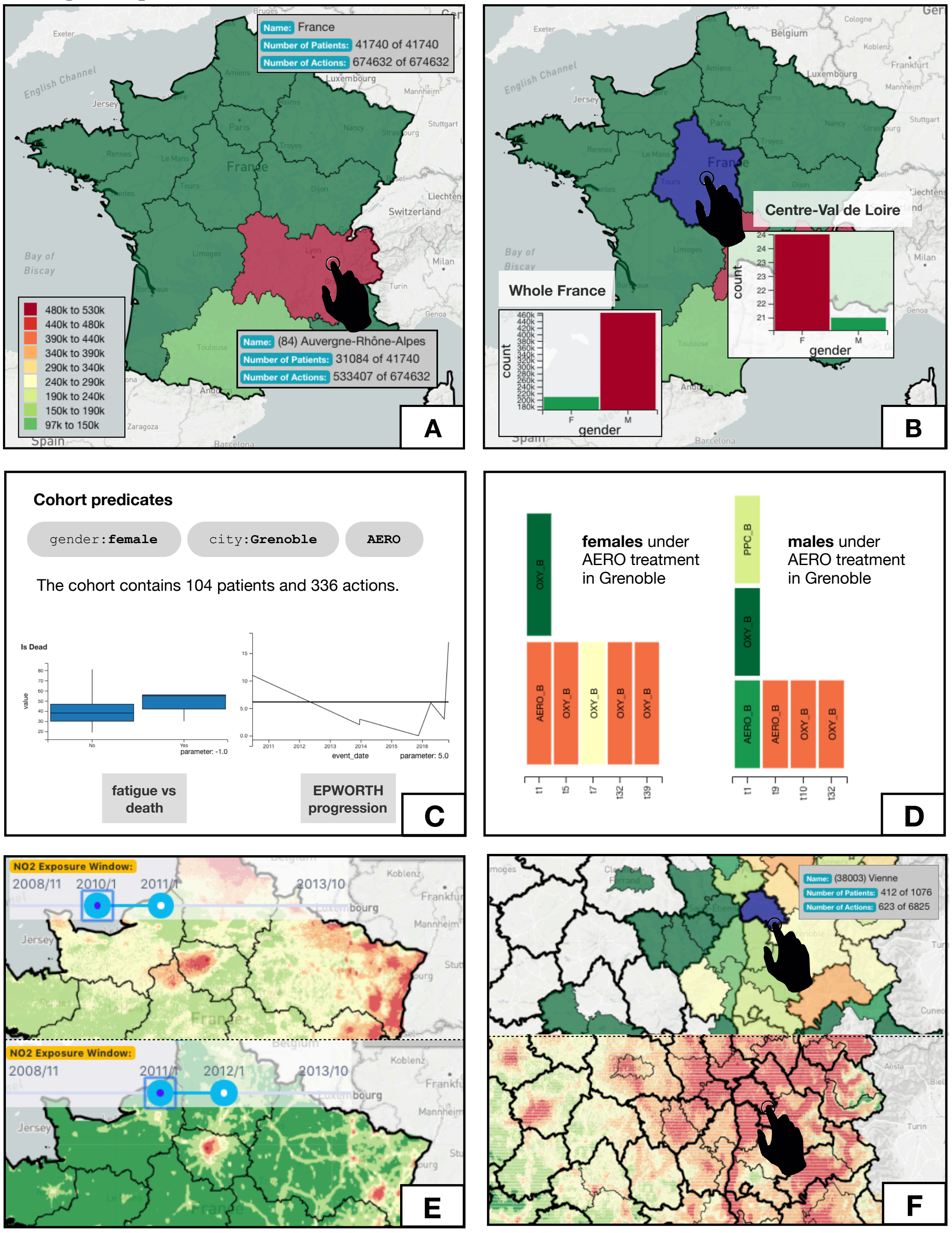
We implement count queries and on-the-fly aggregations (which are materialized only at the execution time to save memory) by query algorithms that operate directly on the pivot lists. To enable quantile queries, QDS augments each entry in the pivot lists with a compressed representation of a distribution function based on a non-parametric distribution modeling technique called t-digest (DUNNING; ERTL,). We store such representation as a payload of numeric dimensions, which also support on-the-fly aggregations and merging of distribution functions. QDS supports the selection of predicates for cohort formation. On-the-fly aggregations build a data view which constitutes a cohort. Hence cohorts are directly indexed in QDS.

In COVIZ, aggregated values of cohorts are not limited to averages but distributions within different quantiles. For instance, instead of indexing a single average value of BMI marker for the cohort of females under AERO treatment in Grenoble, QDS stores its quantiles. As a result, all aggregations can be computed on-the-fly, such as average, quantile, max, and min. Our experiments in (Pahins et al., 2017) shows that spatial extensions of PostgreSQL, SQLite, and MonetDB fail to render an interactive performance for filtering and combining spatial, temporal, and categorical dimensions. QDS renders all kinds of filters with an average delay of $40ms$, enabling exploratory cohort formation.

5.3.3 Cohort exploration

Once a cohort is formed, the medical expert expects to examine "what happened to its members" by exploring the cohort. This question relates to finding and conveying the health trajectory of a cohort in a human-understandable way. The cohort trajectory helps medical experts to generate hypotheses on the health evolution of the cohort's members. Obtaining a readable and succinct trajectory is challenging because cohorts often consist of hundreds of patients whose medical events are of various types and occur at different points in time. An ideal health trajectory should describe a single end-to-end storyline for the cohort and be limited to what matters the most in the cohort (i.e., be succinct). In (OMIDVAR-TEHRANI; AMER-YAHIA; LAKSHMANAN, 2018), we developed an algorithm which iterates over all pairs of patients in the cohort to verify if there is a common match between their health trajectories. Given the sequential nature of medical events, matches are identified using Needleman-Wunsch sequence matching algorithm (POLYANOVSKY; ROYTBERG; TUMANYAN, 2011). Highly frequent events will then be reported in the cohort trajectory. In our user study with medical experts, the representativity, usefulness, and novelty of cohort trajectories were evaluated, where average scores of 4.2, 4.5, and 3.7 (out of 5) were obtained, respectively (OMIDVAR-TEHRANI; AMER-YAHIA; LAKSHMANAN, 2018).

Figure 5.3: Tasks in COVIZ: cohort formation (A-B), cohort comparison (C-D), sense-making with pollution data (E-F).



Moreover, cohorts can be compared using their trajectories. For instance, comparing the cohorts of patients in urban and rural regions of France reveals similarities and differences between their health evolution. In our user study, we also asked the medical experts to evaluate the representativity, usefulness, and novelty of cohort comparisons using their trajectories, and we obtained average scores of 4.03, 4.67, and 4.35, respectively.

5.4 Interface

We present different features of the COVIZ interface using a visualiz-ation-driven scenario[1] (see Figure 5.3). We consider a medical expert who is interested to obtain insights by visual inspection of the health-care data. Typically, she needs first to acquire an overall understanding of the data. Then she seeks to form interesting cohorts and explore them. Last, she seeks to make sense of her observations by leveraging the pollution data.

Observing the big picture. QDS enables an immediate materialization of the big picture to depict general trends in the health-care data. This helps experts make more informed decisions when forming cohorts. Figure 5.3-A visualizes this big picture for $41,740$ patients and $674,632$ of their events. One can easily notice that the geographical distribution of the data is biased towards the Auvergne-Rhône-Alpes region, where the headquarters of our medical partner are located. A mouse-hover on this region reveals that it contains $31,084$ patients. Beyond the big picture, COVIZ provides histograms to examine distributions of different dimensions of patients' health. Figure 5.3-B shows that while 90% of patients are male in all of France (in our data), 80% of the sub-population in the region of Centre-Val de Loire is female. Histograms in COVIZ are inter-connected, i.e., a filter on one histogram updates all other statistics instantaneously.

Cohort formation. Visual filters can be used to form a cohort, e.g., "females under AERO treatment in Grenoble" (Figure 5.3-F). This example cohort contains 104 patients with 336 events. The system will then show a series of statistics for the selected cohort in an efficient manner. For instance, we observe that the higher values of the fatigue marker in the cohort relates to death. We also observe that the progression of the sleepiness marker decreased until late 2015 and then it increased again.

Cohort exploration. Experts can examine the health evolution of cohort's members using cohort trajectories. Figure 5.3-F shows the cohort trajectory of females under AERO treatment in Grenoble. The trajectory shows that the cohort's members started their treatment with AERO and OXY. Then they had a series of OXY treatments in four consecutive months. Additionally, multiple cohorts can be formed and compared visually. Figure 5.3-

[1] *The examples in Section 5.4 are meant to show various functionalities and potentials of COVIZ and are not necessarily significant clinical-wise. An in-depth medical analysis is needed to turn those observations into actionable insights.*

H shows the cohort trajectory of males under AERO treatment in Grenoble. We observe that while the female cohort systematically received AERO right after their admission to the hospital, the male cohort started receiving it only months later.

Sensemaking with pollution data. COVIZ uses another instance of QDS for pollution data to enable an interactive exploration of that data over different regions and in different granularities. In Figure 5.3-I top, we set the time window (i.e., filtering the temporal dimension) to 2010-2011 and we immediately observe that the north-eastern region of France (région Grand-Est), was highly polluted during that time (with the NO2 pollutant). However, we can also observe that this effect is temporary, as is shown in Figure 5.3-I bottom, where the volume of NO2 is noticeably lower in the period of 2011-2012. Pollution data can also be used to interpret some observations in the health data. For instance, Figure 5.3-J top shows that the variance of the sleepiness marker in the province of Vienne (south-east of France) is higher than usual in the year 2010. Figure 5.3-J bottom shows the pollution data layer on the same region and shows that Vienne was highly polluted then, potentially justifying heterogeneous values of the sleepiness marker. This process identifies a novel hypothesis (e.g. impact of air pollution on sleepiness) that is worth a future in-depth investigation.

5.5 Demonstration Scenarios

We describe two scenarios that the demo attendees can perform on COVIZ during the demo session.

Ecological scenario. In ecological studies, the unit of observation is a cohort and the aim is to analyze the collective behavior of cohort's members. Measurements such as disease rates and exposures are taken for a series of cohorts and then their relation is examined (COGGON; BARKER; ROSE, 2003). A common practice is to compare a pair of cohorts which differ only in one dimension, referred to as contrast cohorts (ELM et al., 2007). This enables medical experts to focus on that dimension and ignore the effect of confounding factors. Demo attendees will be able to test this feature and generate hypotheses on differences between cohorts. For instance, they can verify the adoption of a specific treatment, e.g., OXY, for different genders. They form cohorts of males and females and filter treatments to keep only OXY. Then they can verify the distribution and variability of different markers for those two cohorts using different aggregation modes (average, variance, quantile). They can also compare their trajectories to check if there is a significant difference between the times when the two cohorts received a treatment. Moreover, they can check other treatments which are administered by one cohort but not the other. These observations will enable them to generate hypotheses on the difference in treatment administration for contrast cohorts of interest.

Case cross-over scenario. In environmental epidemiology, a cohort is often compared with its past (usually 2 to 5 previous days) to determine its health evolution, called case cross-over study (FUNG et al., 2003). Demo attendees can form their cohort and investigate its health trajectory in different time windows. For each time window, they can also verify the amount of air pollution for different pollutants. Attendees will be able to generate various hypotheses on the relationship between pollution and the health evolution of their cohort.

6 CONCLUSIONS AND DISCUSSION

This thesis introduces three (Chapters 3, 4 and Appendix A) run-time and memory efficient data cube solutions to answer queries from interactive visualization tools that explore and analyzes large multidimensional datasets by exploiting the *pivot* concept prosed by the author (see Chapter 2). The pivot concept showed that (i) is possible to represent hierarchical and flat data structures using an optimized schema that is stored in a linear fashion way, and (ii) demonstrated that this leads to memory savings over other data cube visualization proposals, enabling researchers to develop richer and seamless interactive visualization tools.

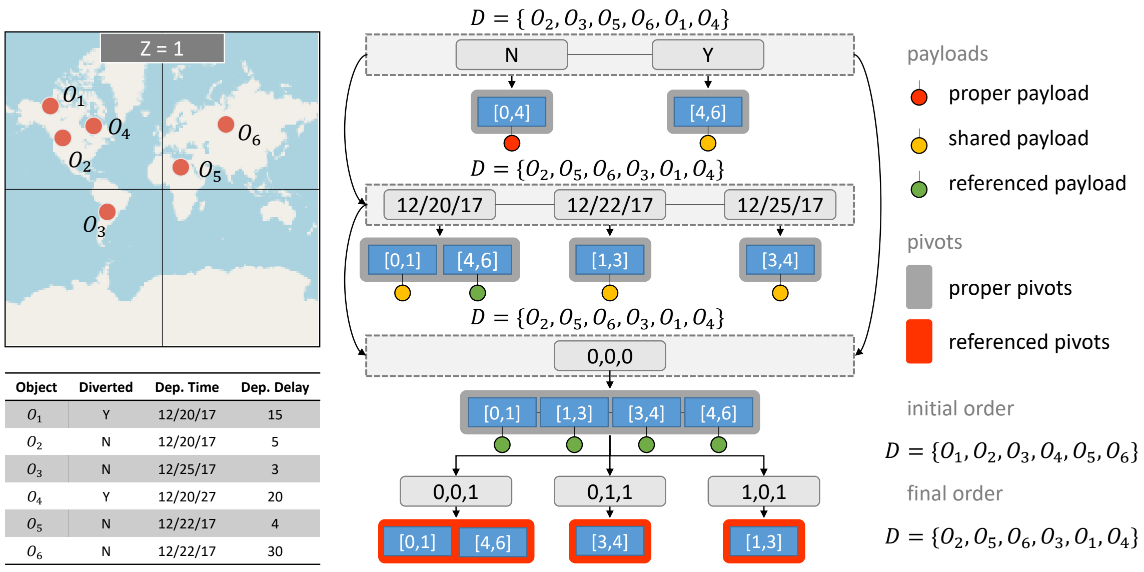
Throughout the publication of the papers of this thesis, the proposed data cubes became more complex and powerful by supporting, at first, counting queries (see Chapter 3), to the representation of approximated values for moment statistic (see Chapter 4). This improvement makes data cubes widely applicable for the analysis of large spatiotemporal datasets. These capabilities open a large number of opportunities to design novel visual encodings and techniques and that the ones explored by the author are just the beginning of a large path to follow. Based on that, it is possible to define two theoretical requirements that an analytical method must implement to be integrated into the data cube infrastructure built by the author: (i) serializable representation and (ii) mergeable disjoint results.

(i) Serializable Representation. Serialization is the ability to convert an object into a stream of bytes to store it to memory. QDS took advantage of this technique and proposed *payload* storing along with *pivot* representation, as in Figure 6.1. As discussed in Appendix C, a *pivot* represents an interval $[i_0, i_1]$, where i_0 and i_1 are the initial and final markers within a given array, and express a list of elements with a common semantic value, such as the same spatial region, categorical or temporal attributes. It can be interpreted as a subset of elements. Along with *pivots*, QDS stores *payloads*, which are objects converted into streams of bytes that represent results of analytical methods over a subset of elements. Any serializable object can be stored by QDS's indexing schema, which employs an aggressive method of identifying *pivots* and *payloads* of redundant data over the data cube, effectively reducing memory usage of serializable analytical methods representation.

(ii) Mergeable Disjoint Results. There are two main technical requirements of QDS that demand mergeable disjoint results. The first requirement originates from its data cube approach. QDS stores only the minimum space of precomputed hierarchical aggregations to be able to recover the combinations of all data dimensions in an n-dimensional lattice, which proves to be effective in reducing the memory usage when compared against competing datacube methods. The used data cube approach leads to singular query algorithm. The general idea of QDS's query algorithm is to select *pivots* across dimensions, find the intersection and, then, summarize or group the result. The algorithm is composed of three steps executed in sequence: *selection*, *intersection* and *aggregation* (see Chapter 3 for the definition of each step). The second technical requirement to mergeable disjoint results originates from the last step of the algorithm, *aggregation*, which perform a series of merge operations of intermediate *pivots* and *payloads* to then group elements by similarity and produce a compact representation of the result.

There are various interesting research paths to improve the analytical capabilities of the author's data cube proposals. QDS's theoretical requirements to integrate novel analytical capabilities are vastly satisfied by methods in the context of streaming and mobile data analysis. The following sections discuss two novel solutions to integrate previously unrelated methods to the data cubes context.

Figure 6.1: Overview of the author's QDS data cube proposal. It employs a novel data cube indexing that reduce memory usage of previous methods and introduces p-digest (de Lara Pahins; Ferreira; Comba, 2019), an efficient quantile sketch that follows the theoretical requirements that an analytical method must implement to be integrated into the data cube infrastructure built by the author.



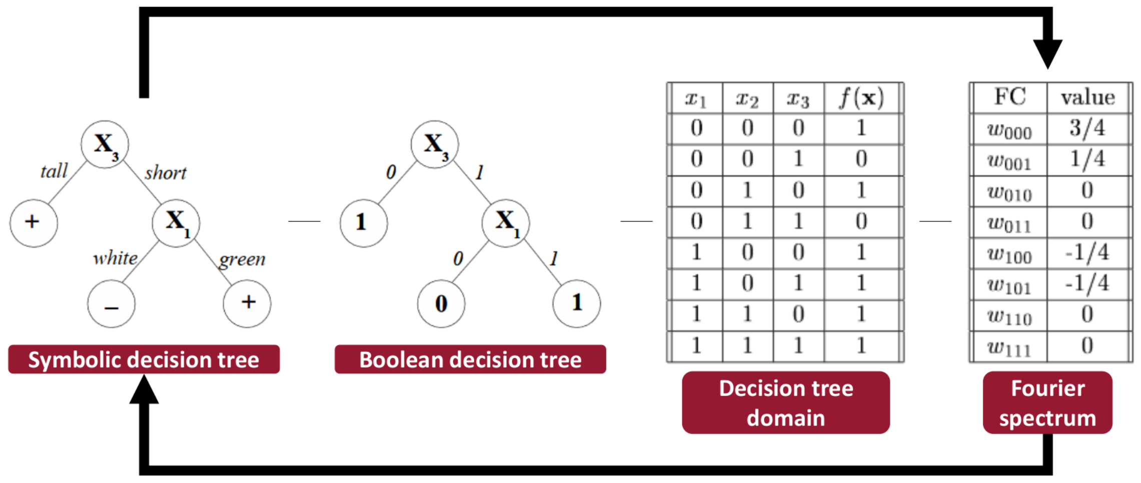
Object	Diverted	Dep. Time	Dep. Delay
O_1	Y	12/20/17	15
O_2	N	12/20/17	5
O_3	N	12/25/17	3
O_4	Y	12/20/27	20
O_5	N	12/22/17	4
O_6	N	12/22/17	30

6.1 A Fourier Spectrum-Based Approach to Represent the Decision Tree Classifier and Regression Method

The analytical capabilities of QDS can be improved by integrating the weel well-known decision tree classifier and regression method. The typical complexity of most used decision tree learning algorithms, e.g., ID3, C4.5 or CART (SINGH; GUPTA, 2014), is $O(n_{features}n^2_{samples}\log(n_{samples}))$ (WITTEN; FRANK; HALL, 2011), which is not appropriated to real-time analysis scenarios. The precomputation of decision trees for the hierarchical aggregation of all data dimensions in a n-dimensional lattice, following the data cube concept, can be a solution for that.

Implementation. Kargupta and Park proposed a Fourier spectrum-based representation of decision trees in the context of mining a data stream in mobile environments (KARGUPTA; PARK, 2004) that is capable of merging incrementally modifying decision trees, as well as aggregation of multiple trees frequently generated by ensemble-based methods. They provide the theory to compute the Fourier spectrum of a decision tree and the decision tree from its Fourier spectrum, which is a valuable source to design algorithms suitable to data cubes, as observed in Figure 6.2. A *proof-of-concept* implementation was proposed by the author of this thesis in Appendix C, and demonstrates that is feasible to achieve a Fourier spectrum-based representation of decision trees that is well suited to the *quantile sketches* of QDS.

Figure 6.2: Overview of the process of computing the Fourier spectrum of a decision tree and vice-versa. Adapted from (KARGUPTA; PARK, 2004). A *proof-of-concept* implementation was proposed by the author of this thesis in Appendix C.



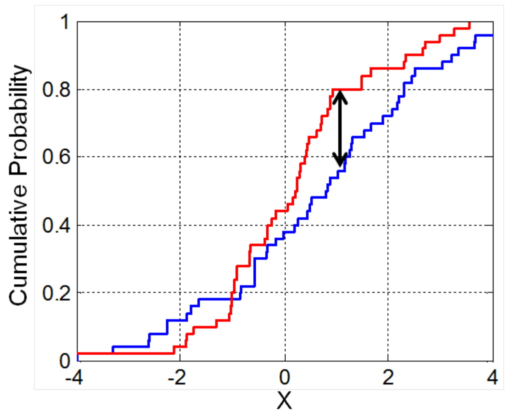
x_1	x_2	x_3	$f(\mathbf{x})$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

FC	value
w_{000}	3/4
w_{001}	1/4
w_{010}	0
w_{011}	0
w_{100}	-1/4
w_{101}	-1/4
w_{110}	0
w_{111}	0

6.2 A Clustering Technique

QDS is a data structure that integrates *quantile sketches* (data sketches that support queries of quantile and *cdf* estimation) into a data cube method. Based on that, a distance between two quantile sketches can be calculated using the two-sample Kolmogorov-Smirnov (KS) test (LOPES, 2011). The two-sample KS test measures the difference between two one-dimensional probability distributions, as observed in Figure 6.3. Following the ideas from the classical K-means algorithm, it's possible to design a clustering algorithm that uses the KS distance as metric to compare features distributions.

Figure 6.3: Illustration of the two-sample Kolmogorov–Smirnov test. The black arrow is the KS distance between two one-dimensional probability distributions.

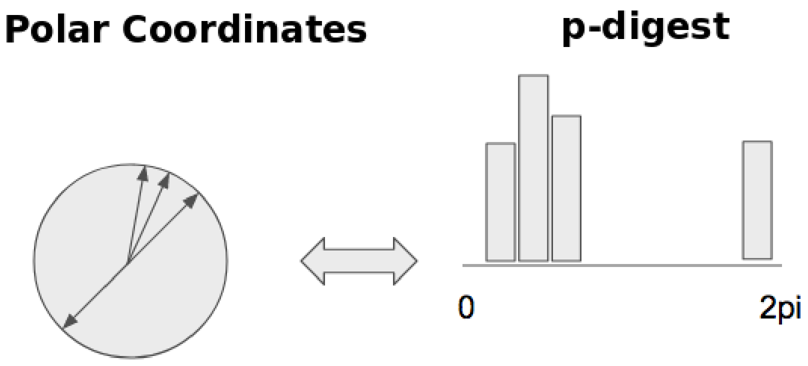


As well as the classic K-means algorithm, the *QDS* clustering algorithm starts by choosing a centroid value for each cluster. In this initialization step, the QDS algorithm randomly choose an initial cluster. The centroids of the following initial clusters are chosen so as to maximize the KS distance of the features distributions. For example, if the algorithm is executed with $K = 3$, the centroid of the cluster C_1 will be chosen randomly in the dataset. The cluster C_2 will be chosen so as to maximize the KS distance from the centroid C_1. The last centroid C_3, will be chosen in order to maximize the KS distance for all previously selected centroids, that is, $\{C_1, C_2\}$. After that, the algorithm iteratively performs three steps: (i) Find the KS distances for all p-digest modeled features distributions between each data instance and centroids of all the clusters; (ii) Assign the data instances to the cluster of the centroid with nearest KS distance; (iii) Calculate new centroid values based on the mean values of the features distributions of all the data instances from the corresponding cluster.

The modeling capabilities of p-digest were discussed in Chapter 4, but the clustering algorithm of QDS can also use p-digest as a way to represent a set of directions.

Based on the polar coordinate system, where a point is represented by a radial distance ρ and an angle ϕ, it is possible to use p-digest to model the distribution of a set of directions $\{\phi_1, \phi_2, ..., \phi_{n-1}, \phi_n\}$, as shown in Figure 6.4. To perform the clustering, the QDS algorithm compares the KS distances of multiple features distributions, each stored in its own p-digest. To find the optimal degree of relevance for each feature in this linear combination $\{Feature_1 * Weight_1, Feature_2 * Weight_2, ..., Feature_k * Weight_k\}$, one can use, for instance, the well known Stochastic Gradient Descent (SGD) method to optimize the objective function defined as the convergence score.

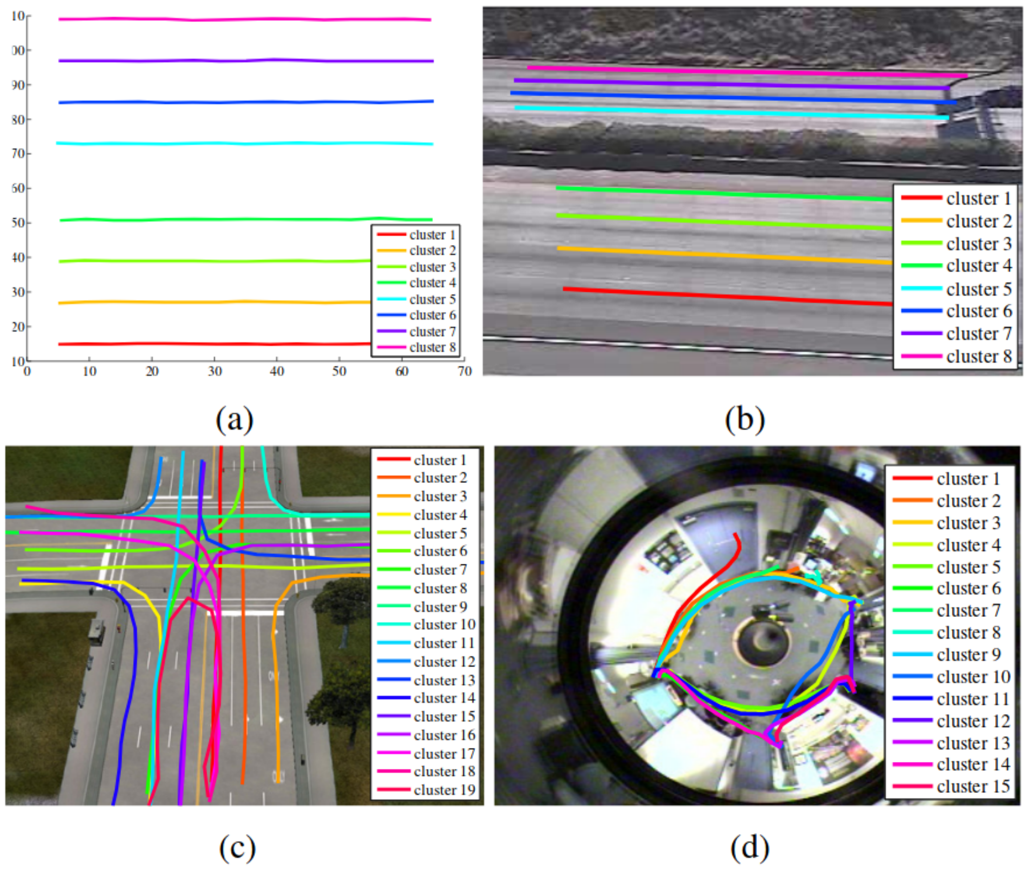
Figure 6.4: p-digest enables the representation of a set of directions by modeling it's distribution into the range (0 to 2π). By using the KS distance, the clustering algorithm can measure the similarity between distributions that represent directions.



The convergence of the clustering algorithm is dictated by two thresholds: (i) number of iterations, and (ii) tolerance value (tol). The *number of iterations* is a hard limit to stop the algorithm from running when it reach N iterations. This hard threshold can be very useful for experiments, such as in cases where it is necessary to study the behavior of clustering to assist in the *feature engineering*, however, when used individually, it is inefficient to detect convergence. The second threshold, the tolerance value tol, stops the clustering algorithm if the score or cumulative error does not improve by at least tol for N consecutive iterations. The cumulative error for the QDS clustering algorithm is the squared sum of KS distances of all elements in their corresponding clusters. Note that tol does not guarantee that the clustering algorithm will find the optimal centroids, neither stops the algorithm from finding a local optimum, since this also depends on the initial choice made in the initialization step of the algorithm, which is random.

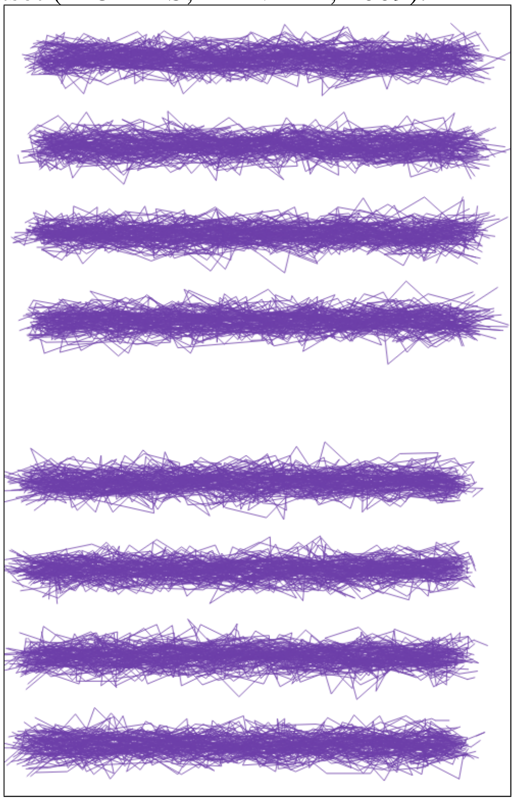
Experiments. Initial experiments were conducted with the dataset provided by Morris and Trivedi (MORRIS; TRIVEDI, 2009), as observed in Figure 6.5. This dataset is composed of a series of trajectories with distinct properties, direction and velocity, and helps to understand the behavior of the clustering algorithm in simplified situations, as in cases of parallel trajectories (Figures 6.5-a and 6.5-b), since it is expected that the algorithm will be able to easily discover clusters.

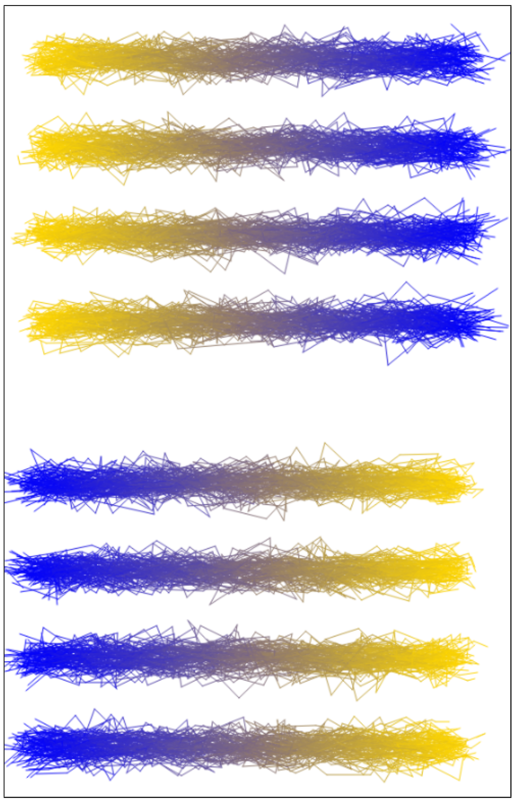
Figure 6.5: Collection of simulated scenes and datasets with varying properties. (a) Trajectories obtained on a four lane highway with traffic in both directions. (b) Trajectories obtained by visual tracking of vehicles from a highway. (c) Trajectories obtained from a four traffic intersection. (d) Trajectories obtained by tracking a laboratory. Adapted from (MORRIS; TRIVEDI, 2009).

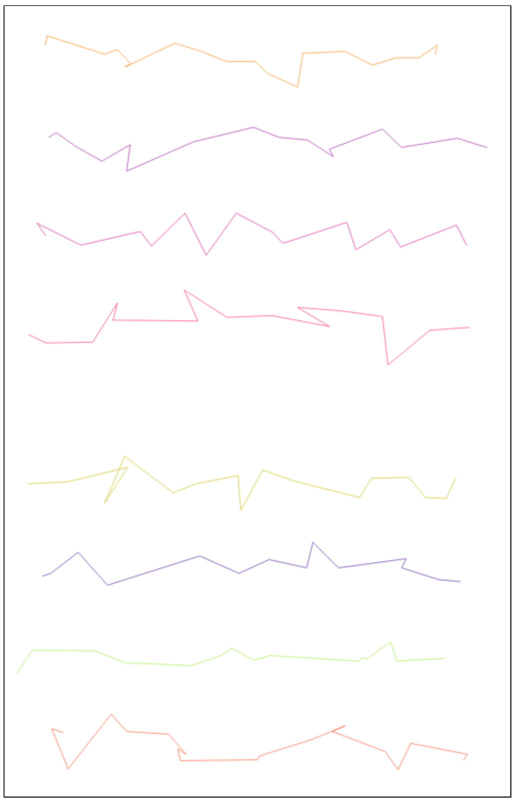


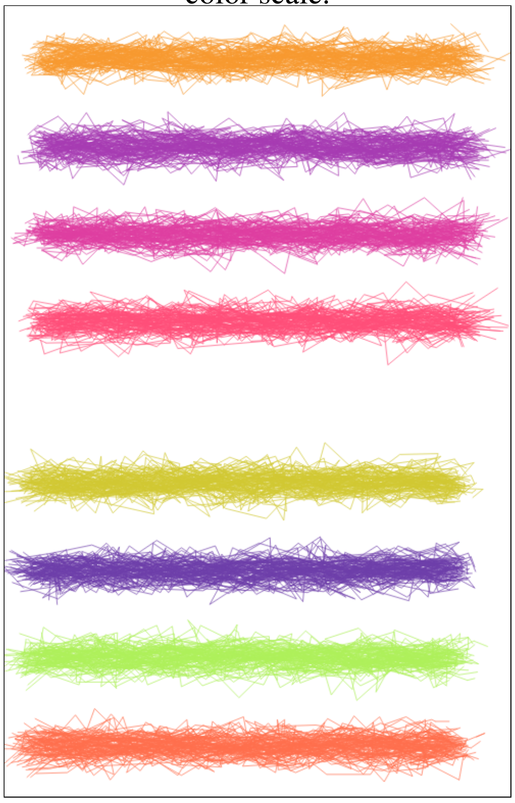
In the first experiment, we used the trajectories observed in Figures 6.5-a. For this, all points of the trajectories are indexed using the special index of the QDS, with the association of *speed* values. Note that in Figure 6.6-a and 6.6-b, the upper set of trajectories have right-to-left sense, while the lower set of trajectories has the opposite direction. The clustering algorithm is able to detect this characteristic by modeling the distribution of the direction of these trajectories, as observed in Figure 6.4. To differentiate upper and lower sets of trajectories, the clustering algorithm takes into account the spatial index of QDS. For each trajectory, the between the path and the center of all clusters is measured, associating it to the cluster of smaller distance. This process is performed for all points of

Figure 6.6: Visualization of 8 clusters generated by QDS using Morris et al. dataset (MORRIS; TRIVEDI, 2009).


(a) Overview of the data.


(b) Visualization of the trajectories using a blue (begin) to orange (end) color scale.


(c) Initial $k = 8$ clusters.


(d) After $N = 2$ iterations.

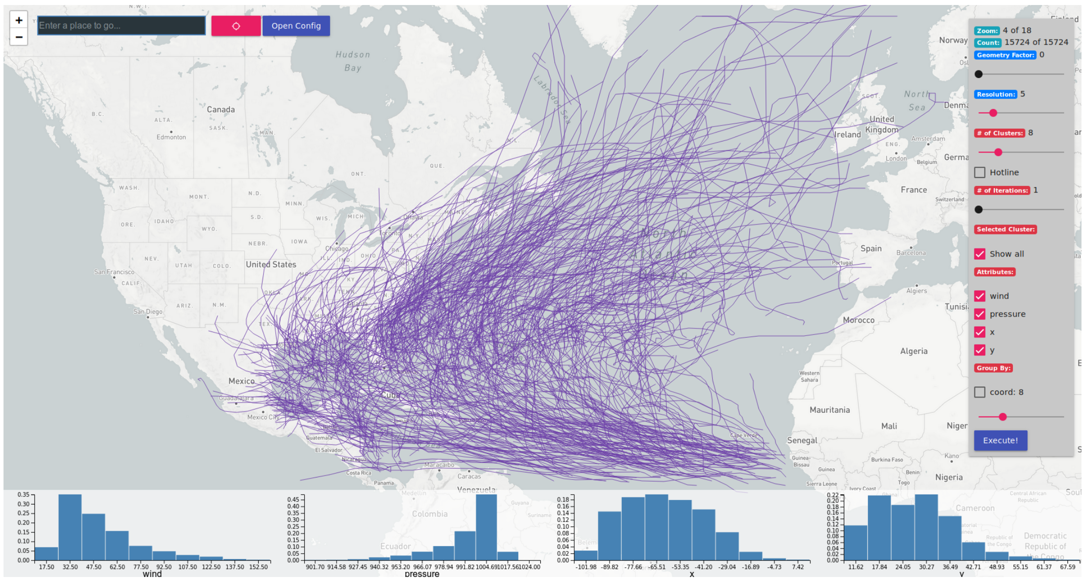
a trajectory, averaging the final result. Due to the simplicity of the problem, the clustering algorithm discovered the correct clusters after only two iterations (Figure 6.6-d).

The second experiment was conducted with the HURDAT2 dataset (LANDSEA; FRANKLIN, 2013), which combines the trajectories of all known tropical and subtropical cyclones from 1851 to 2017. Each trajectory has a six-hourly information on the location, maximum winds, central pressure, and size of the cyclone. The main objective of this experiment is to test if the clustering algorithm is able to use the combination of more complex features to produce meaningful clusters. As in the previous experiment, all points of the trajectories were indexed in the spatial dimension of the *QDS*, as well as their location, wind and pressure features were modeled by p-digest, as observed in the bottom of Figure 6.7-a. Through the linear combination of the KS distances of each feature modeled by p-digest (location, wind, and pressure), the QDS clustering algorithm was able to correctly form different sets of cyclones with similar characteristics, with results very close to the work of Singh et al. (SINGH; GUPTA, 2014), even though they are not necessarily spacially close. This experiment helps to demonstrate that is feasible to achieve an accurate clustering method that is based on the *quantile sketches* of QDS.

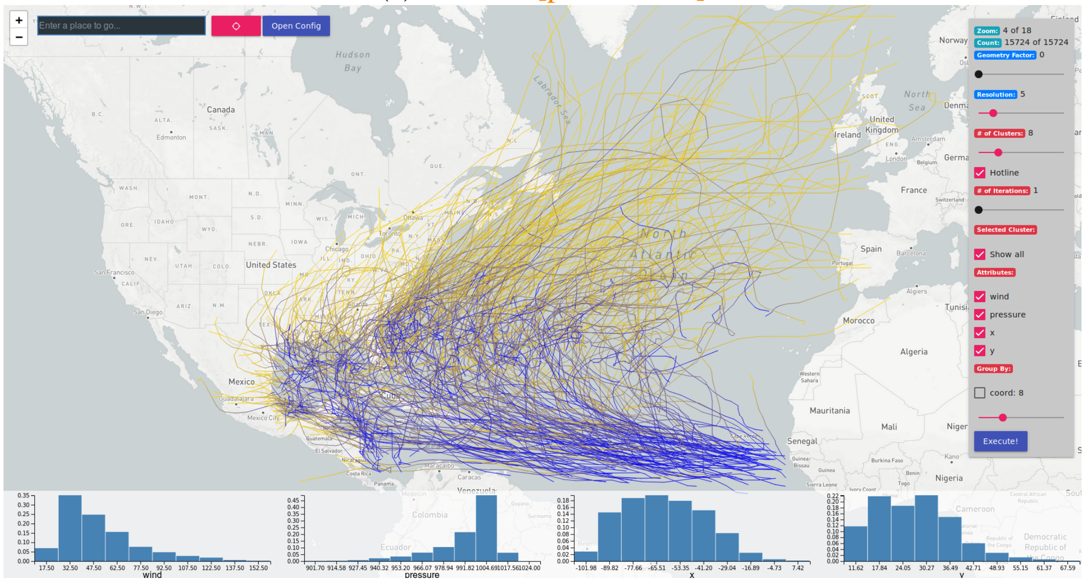
Conclusion. The real-time exploration and analysis of big data are one of the primary desires of visualization practitioners and data scientists. This thesis proposed five contributions to this area. A technique to avoid the large memory footprint commonly used to accelerate data queries (Chapter 2), three data cubes solutions that take advantage of the proposed memory footprint reduction technique in the context of static datasets (Chapters 3, 4, and Appendix A), and a solution designed to support exploration and analysis of streaming data (Appendix B).

The opportunities for improvement of the current analytical capabilities of the solutions proposed in this thesis are identified and were discussed in Sections 6.1 and 6.2. The discussed improvements help to highlight the relevance of the collection of papers presented in this thesis, since, in majority, these papers involve state-of-the-art analytical methods of big data in the context of data cubes, data structures that typically lead to the exponential memory problem, and introduce methods that achieve low latency and memory usage by taking advantage of the infrastructure built around the author's work.

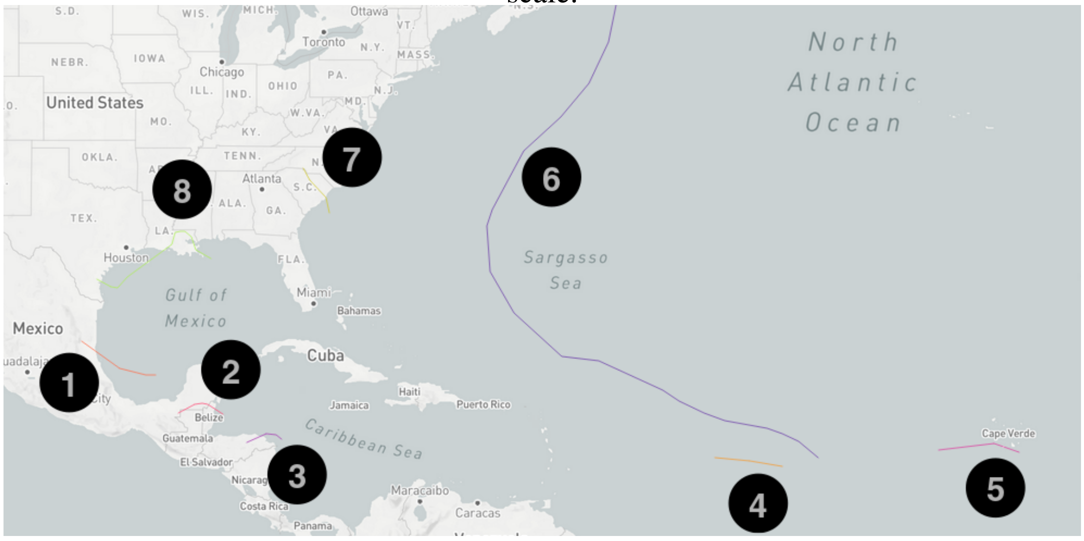
Figure 6.7: Overview of the *HURDAT2* dataset. The data consists of tracking information of all known tropical and subtropical cyclones from 1851 to 2017 (LANDSEA; FRANKLIN, 2013).


Zoom: 4 of 18
Count: 15724 of 15724
Geometry Factor: 0
Resolution: 5
# of Clusters: 8
Hotline
# of Iterations: 1
Selected Cluster:
Show all
Attributes:
wind
pressure
x
y
Group By:
coord: 8

(a) Review: [placeholder]

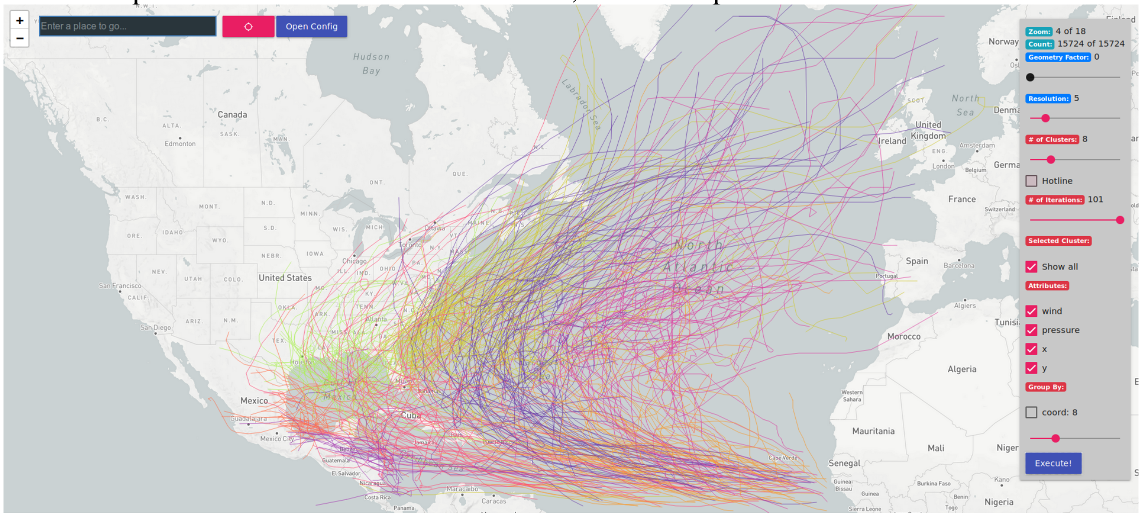

Zoom: 4 of 18
Count: 15724 of 15724
Geometry Factor: 0
Resolution: 5
# of Clusters: 8
Hotline
# of Iterations: 1
Selected Cluster:
Show all
Attributes:
wind
pressure
x
y
Group By:
coord: 8

(b) Visualization of the hurricanes paths using a blue (begin) to orange (end) color scale.



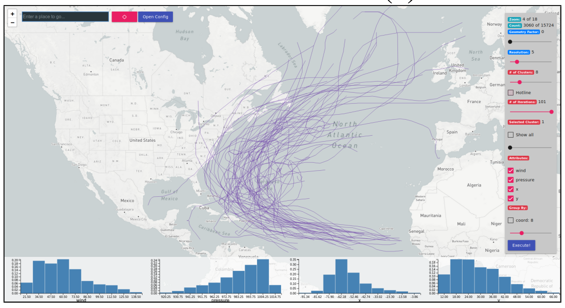
(c) Initial $k = 8$ clusters of QDS Clustering method.

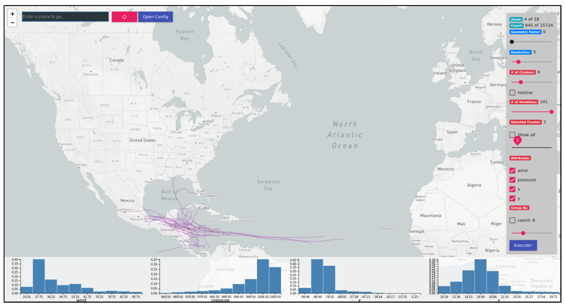
Figure 6.8: Visualization of 8 clusters generated by QDS using HURDAT2 dataset. Each hurricane path is clustered based on location, wind and pressure features.

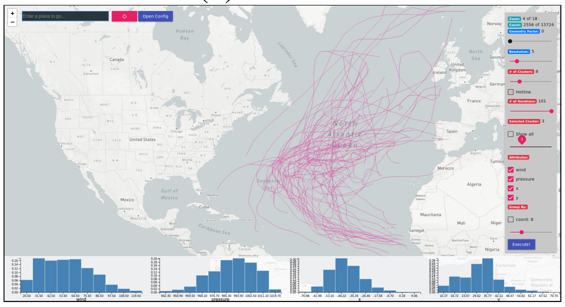


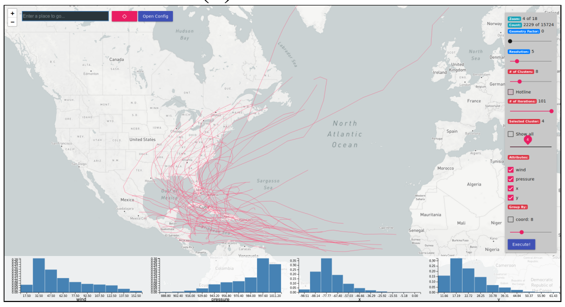
Zoom: 4 of 18
Count: 15724 of 15724
Geometry Factor: 0
Resolution: 5
# of Clusters: 8
Hotline
# of Iterations: 101
Selected Cluster:
Show all
Attributes:
wind
pressure
x
y
Group By:
coord: 8
Execute!

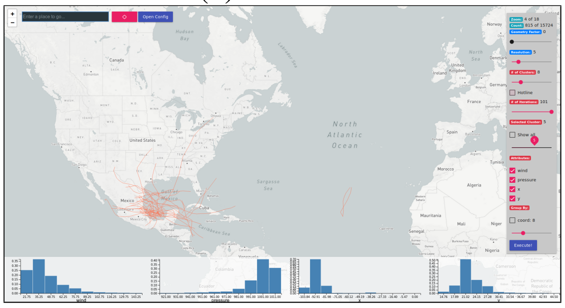
(a) Collection of all clusters.

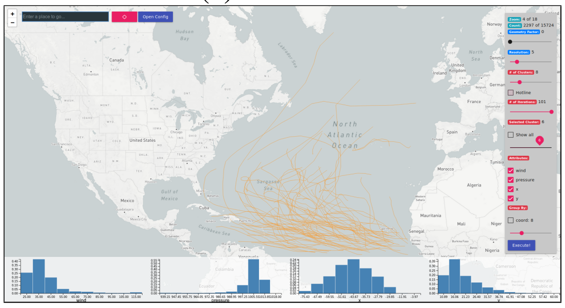

(b) Cluster 1.

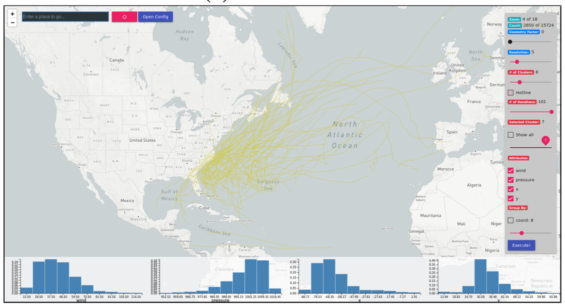

(c) Cluster 2.

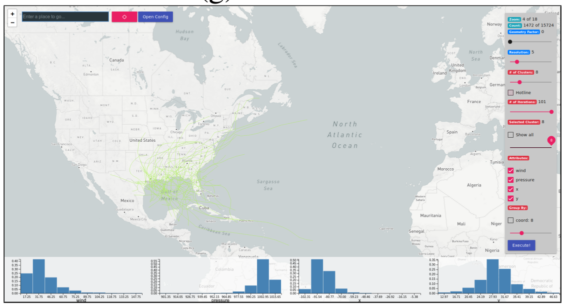

(d) Cluster 3.


(e) Cluster 4.


(f) Cluster 5.


(g) Cluster 6.


(h) Cluster 7.


(i) Cluster 8.

REFERENCES

AGAFONKIN, V. **Leaflet - a JavaScript Library for Mobile-Friendly Interactive Maps**. 2014. <http://leafletjs.com/>.

AGARWAL, P. K. et al. Mergeable Summaries. **ACM Transactions on Database Systems**, ACM, v. 38, n. 4, p. 26, 2013.

AGARWAL, S. et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In: **Proceedings of the 8th ACM European Conference on Computer Systems**. [S.l.]: ACM, 2013. (EuroSys '13), p. 29–42. ISBN 978-1-4503-1994-2.

AGRAWAL, R. et al. Challenges and opportunities with big data visualization. In: **Proc. of the 7th International Conference on Management of Computational and Collective intelligence in Digital EcoSystems**. [S.l.]: ACM, 2015. (MEDES '15), p. 169–173. ISBN 978-1-4503-3480-8.

Airlines for America. **U.S. Passenger Carrier Delay Costs**. <https://tinyurl.com/ycmxgcoy>. Accessed: 2018-07-18.

AL, M. B. et. Chronic effects of air pollution on lung function after lung transplantation in the systems prediction of chronic lung allograft dysfunction (sysclad) study. In: EUROPEAN RESPIRATORY JOURNAL. [S.l.], 2016.

AMATO, F. et al. Semtree: An index for supporting semantic retrieval of documents. In: IEEE. **Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on**. [S.l.], 2015. p. 62–67.

AMER-YAHIA, S. et al. Exploration of user groups in vexus. **ICDE demo**, 2018.

American Statistical Association Data Expo. **Airline on-time performance dataset**. 2009. Disponível em: <http://stat-computing.org/dataexpo/2009/>.

APACHE Flink. 2017. <https://flink.apache.org>.

ASSENT, I. et al. The ts-tree: Efficient time series search and retrieval. In: **Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology**. [S.l.]: ACM, 2008. p. 252–263. ISBN 978-1-59593-926-5.

BATTLE, G.; CHANG, R.; STON, M. **Dynamic Prefetching of Data Tiles for Interactive Visualization**. [S.l.], 2015.

BATTLE, L.; CHANG, R.; STONEBRAKER, M. Dynamic Prefetching of Data Tiles for Interactive Visualization. In: ACM. **Proceedings of the 2016 International Conference on Management of Data**. [S.l.], 2016. p. 1363–1375.

BATTLE, L.; STONEBRAKER, M.; CHANG, R. Dynamic reduction of query result sets for interactive visualizaton. In: **Big Data, 2013 IEEE International Conference on**. [S.l.: s.n.], 2013. p. 1–8.

BAYER, R.; MCCREIGHT, E. Organization and maintenance of large ordered indexes. **Acta Informatica**, v. 1, p. 173–189, 1972.

BEN-HAIM, Y.; TOM-TOV, E. A Streaming Parallel Decision Tree Algorithm. **Journal of Machine Learning Research**, v. 11, n. Feb, p. 849–872, 2010.

BENDER, M. A.; DEMAINE, E. D.; FARACH-COLTON, M. Cache-oblivious b-trees. **SIAM J. Comput.**, v. 35, n. 2, p. 341–358, 2005.

BENDER, M. A. et al. **Cache-Oblivious Streaming B-trees**. San Diego, CA, USA: [s.n.], 2007. 81–92 p.

BENDER, M. A.; HU, H. An adaptive packed-memory array. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, v. 32, n. 4, nov. 2007. ISSN 0362-5915.

BENDER, M. A.; HU, H. An adaptive packed-memory array. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, v. 32, n. 4, nov. 2007. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/1292609.1292616>.

BENDER, M. A. M.; DEMAINE, E. D. E. E. D. E.; FARACH-COLTON, M. Cache-Oblivious B-Trees. **SIAM Journal on Computing**, IEEE Comput. Soc, v. 35, n. 2, p. 341–358, jan 2005.

BERN, M.; EPPSTEIN, D.; TENG, S.-H. Parallel construction of quadtrees and quality triangulations. In: ____. **Algorithms and Data Structures: Third Workshop, WADS ’93 Montréal, Canada, August 11–13, 1993 Proc.** Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 188–199.

BINGMANN, T. **STX B+ Tree C++ Template Classes v0.9**. 2013. <https://github.com/bingmann/stx-btree>.

BOOST. **Geometry Index**. 2017. <http://www.boost.org/>.

BOSTOCK, M. **D3.js - Data-Driven Documents**. 2015. <https://d3js.org/>.

BRODAL, G. S.; FAGERBERG, R. **Lower Bounds for External Memory Dictionaries**. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. 546–554 p.

CAMERRA, A. et al. isax 2.0: Indexing and mining one billion time series. In: **Proceedings of the 2010 IEEE International Conference on Data Mining**. [S.l.]: IEEE Computer Society, 2010. p. 58–67. ISBN 978-0-7695-4256-0.

CAO, G. et al. A scalable framework for spatiotemporal analysis of location-based social media data. **Computers, Environment and Urban Systems**, v. 51, p. 70 – 82, 2015. ISSN 0198-9715.

CARPENTER, J.; HEWITT, E. **Cassandra: The Definitive Guide: Distributed Data at Web Scale**. [S.l.]: " O’Reilly Media, Inc.", 2016.

CHAUDHURI, S.; DING, B.; KANDULA, S. Approximate Query Processing: No Silver Bullet. In: ACM. **Proceedings of the 2017 ACM International Conference on Management of Data**. [S.l.], 2017. p. 511–519.

CHAZELLE, B.; GUIBAS, L. Fractional cascading: I. a data structuring technique. **Algorithmica**, Springer-Verlag, v. 1, n. 1-4, p. 133–162, 1986. ISSN 0178-4617.

CHO, E.; MYERS, S. A.; LESKOVEC, J. Friendship and mobility: User movement in location-based social networks. In: **Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. [S.l.]: ACM, 2011. p. 1082–1090. ISBN 978-1-4503-0813-7.

CHO, E.; MYERS, S. A.; LESKOVEC, J. Friendship and mobility: User movement in location-based social networks. In: **Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: ACM, 2011. (KDD '11), p. 1082–1090. ISBN 978-1-4503-0813-7. Disponível em: <http://doi.acm.org/10.1145/2020408.2020579>.

COGGON, D.; BARKER, D.; ROSE, G. **Epidemiology for the uninitiated**. 5. ed. London: BMJ Books, 2003. 73 p.

COOK, C. R.; KIM, D. J. Best sorting algorithm for nearly sorted lists. **Commun. ACM**, ACM, New York, NY, USA, v. 23, n. 11, p. 620–624, nov. 1980. ISSN 0001-0782.

CORMODE, G. et al. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. **Foundations and Trends in Databases**, Now Publishers Inc., v. 4, n. 1–3, p. 1–294, 2012.

CORRELL, M.; GLEICHER, M. Error bars Considered Harmful: Exploring Alternate Encodings for Mean and Error. **IEEE transactions on Visualization and Computer Graphics**, IEEE, v. 20, n. 12, p. 2142–2151, 2014.

CROTTY, A. et al. Vizdom: interactive analytics through pen and touch. **VLDB**, 2015.

de Lara Pahins, C. A.; Ferreira, N.; Comba, J. Real-time exploration of large spatiotemporal datasets based on order statistics. **IEEE Transactions on Visualization and Computer Graphics**, p. 1–1, 2019. ISSN 1077-2626.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In: **OSDI'04: Proceedings Of The 6th Conference On Symposium On Operating Systems Design And Implementation**. [S.l.]: USENIX Association, 2004.

DORAISWAMY, H. et al. Using Topological Analysis to Support Event-guided Exploration in Urban Data. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 20, n. 12, p. 2634–2643, 2014.

DORAISWAMY, H. et al. A gpu-based index to support interactive spatio-temporal queries over historical data. In: **2016 IEEE 32nd International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2016. p. 1086–1097.

DUNNING, T.; ERTL, O. **Computing Extremely Accurate Quantiles Using t-Digests**. <https://github.com/tdunning/t-digest>. Accessed: 2018-07-18.

DUNNING, T.; ERTL, O. **Computing Extremely Accurate Quantiles Using t-Digests**. 2014. <https://github.com/tdunning/t-digest>. Accessed: 2018-07-18.

DURAND, M.; RAFFIN, B.; FAURE, F. **A Packed Memory Array to Keep Moving Particles Sorted**. 2012.

ELM, E. V. et al. The strengthening the reporting of observational studies in epidemiology (strobe) statement: guidelines for reporting observational studies. **PLoS medicine**, Public Library of Science, v. 4, n. 10, p. e296, 2007.

ELMQVIST, N.; FEKETE, J.-D. Hierarchical Aggregation for Information Visualization: Overview, Techniques, and Design Guidelines. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 16, n. 3, p. 439–454, 2010.

ESTIVILL-CASTRO, V.; WOOD, D. A survey of adaptive sorting algorithms. **ACM Comput. Surv.**, v. 24, n. 4, p. 441–476, 1992.

FEKETE, J.-D.; PRIMET, R. Progressive analytics: A computation paradigm for exploratory data analysis. **arXiv preprint arXiv:1607.05162**, 2016.

FEKETE, J.-D. et al. Managing Data for Visual Analytics: Opportunities and Challenges. **IEEE Data Eng. Bull.**, v. 35, n. 3, p. 27–36, 2012.

FELBER, D.; OSTROVSKY, R. A randomized online quantile summary in $\mathcal{O}((1/\varepsilon)\log(1/\varepsilon))$ words. **Theory of Computing**, Theory of Computing, v. 13, n. 14, p. 1–17, 2017.

FENG, W. et al. Streamcube: Hierarchical spatio-temporal hashtag clustering for event exploration over the twitter stream. In: **2015 IEEE 31st International Conference on Data Engineering, ICDE 2015**. [S.l.: s.n.], 2015. p. 1561–1572.

FERNANDES, M. et al. Uncertainty Displays Using Quantile Dotplots or CDFs Improve Transit Decision-Making. In: ACM. **Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems**. [S.l.], 2018. p. 144.

FERREIRA, N.; FISHER, D.; KONIG, A. C. Sample-oriented Task-driven Visualizations: Allowing Users to Make Better, More Confident Decisions. In: ACM. **Proc. Conference on Human Factors in Computing Systems (CHI)**. [S.l.], 2014. p. 571–580.

FISHER, D. et al. Trust me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. In: ACM. **Proc. Conference on Human Factors in Computing Systems (CHI)**. [S.l.], 2012. p. 1673–1682.

FISHER, D. et al. Trust me, i'm partially right: Incremental visualization lets analysts explore large datasets faster. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. [S.l.]: ACM, 2012. (CHI '12), p. 1673–1682. ISBN 978-1-4503-1015-4.

FOX, A. et al. **Spatio-temporal indexing in non-relational distributed databases**. 2013. 291-299 p.

FRAIMAN, R.; MUNIZ, G. Trimmed Means for Functional Data. **Test**, Springer, v. 10, n. 2, p. 419–440, 2001.

FRIGO, M. et al. **Cache-Oblivious Algorithms**. Washington, DC, USA: IEEE Computer Society, 1999. 285– p. (FOCS '99).

FUNG, K. Y. et al. Comparison of time series and case-crossover analyses of air pollution and hospital admission data. **International journal of epidemiology**, Oxford University Press, 2003.

GANI, A. et al. A survey on indexing techniques for big data: taxonomy and performance evaluation. **Knowledge and information systems**, Springer, v. 46, n. 2, p. 241–284, 2016.

GARGANTINI, I. An effective way to represent quadtrees. **Commun. ACM**, ACM, New York, NY, USA, v. 25, n. 12, p. 905–910, dez. 1982. ISSN 0001-0782.

GODFREY, P.; GRYZ, J.; LASEK, P. **Interactive Visualization of Large Data Sets**. [S.l.], 2015.

GOODMAN, J. R. Using cache memory to reduce processor-memory traffic. In: **Proceedings of the 10th Annual International Symposium on Computer Architecture**. [S.l.]: ACM, 1983. p. 124–131. ISBN 0-89791-101-6.

GORO, F. **Timsort**. 2016. <https://github.com/gfx/cpp-TimSort>.

GRAY, J. et al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-tab, and Sub-totals. **Data Mining and Knowledge Discovery**, Springer, v. 1, n. 1, p. 29–53, 1997.

GRAY, J. et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. **Data Mining and Knowledge Discovery**, Kluwer Academic Publishers, v. 1, n. 1, p. 29–53, jan. 1997. ISSN 1384-5810.

GREENWALD, M.; KHANNA, S. Space-efficient online computation of quantile summaries. **SIGMOD Records**, ACM, New York, NY, USA, v. 30, n. 2, p. 58–66, maio 2001. ISSN 0163-5808.

GUPTA, D.; SIDDIQUI, S. Big data implementation and visualization. In: **Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on**. [S.l.: s.n.], 2014. p. 1–10. ISSN 2347-9337.

GUTTMAN, A. **R-trees: a dynamic index structure for spatial searching**. [S.l.]: ACM, 1984. v. 14.

HAKLAY, M. M.; WEBER, P. Openstreetmap: User-generated street maps. **IEEE Pervasive Computing**, IEEE Educational Activities Department, v. 7, n. 4, p. 12–18, out. 2008. ISSN 1536-1268.

HELLERSTEIN, J. M.; HAAS, P. J.; WANG, H. J. Online aggregation. **ACM SIGMOD Record**, ACM, v. 26, n. 2, p. 171–182, jun. 1997. ISSN 0163-5808.

HUANG, X. et al. Trajgraph: A graph-based visual analytics approach to studying urban network centralities using taxi trajectory data. **IEEE Transactions on Visualization and Computer Graphics**, v. 22, n. 1, p. 160–169, Jan 2016. ISSN 1077-2626.

IDREOS, S.; PAPAEMMANOUIL, O.; CHAUDHURI, S. Overview of data exploration techniques. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2015. (SIGMOD '15), p. 277–281. ISBN 978-1-4503-2758-9.

IM, J.-F.; VILLEGAS, F. G.; MCGUFFIN, M. J. Visreduce: Fast and responsive incremental information visualization of large datasets. In: **2013 IEEE International Conference on Big Data**. [S.l.]: IEEE, 2013. p. 25–32.

ITAI, A.; KONHEIM, A. G.; RODEH, M. **A Sparse Table Implementation of Priority Queues**. 1981. 417–431 p.

JIANG, X. et al. Large-scale taxi o/d visual analytics for understanding metropolitan human movement patterns. **J. Vis.**, Springer-Verlag New York, Inc., v. 18, n. 2, p. 185–200, maio 2015. ISSN 1343-8875.

JO, J. et al. Swifttuna: Responsive and incremental visual exploration of large-scale multidimensional data. In: **Proc. Pacific Visualization Symposium (PacificVis)**. [S.l.: s.n.], 2017. p. 131–140.

JOHNSON, C. R.; SANDERSON, A. R. A Next Step: Visualizing Errors and Uncertainty. **IEEE Computer Graphics and Applications**, v. 23, n. 5, p. 6–10, Sept 2003. ISSN 0272-1716.

JUGEL, U. et al. M4: A visualization-oriented time series data aggregation. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 10, p. 797–808, jun. 2014. ISSN 2150-8097.

JUGEL, U. et al. Vdda: Automatic visualization-driven data aggregation in relational databases. **The VLDB Journal**, Springer-Verlag New York, Inc., v. 25, n. 1, p. 53–77, fev. 2016. ISSN 1066-8888.

KAMAT, N. et al. Distributed and interactive cube exploration. In: **Data Engineering (ICDE), 2014 IEEE 30th International Conference on**. [S.l.: s.n.], 2014. p. 472–483.

KAMAT, N.; NANDI, A. A Session-Based Approach to Fast-But-Approximate Interactive Data Cube Exploration. **ACM Transactions on Knowledge Discovery from Data**, ACM, New York, NY, USA, v. 12, n. 1, p. 9:1–9:26, fev. 2018. ISSN 1556-4681.

KANDEL, S. et al. Profiler: Integrated statistical analysis and visualization for data quality assessment. In: **Proceedings of the International Working Conference on Advanced Visual Interfaces**. [S.l.]: ACM, 2012. p. 547–554. ISBN 978-1-4503-1287-5.

KARGUPTA, H.; PARK, B. . A fourier spectrum-based approach to represent decision trees for mining data streams in mobile environments. **IEEE Transactions on Knowledge and Data Engineering**, v. 16, n. 2, p. 216–229, Feb 2004. ISSN 1041-4347.

KARNIN, Z.; LANG, K.; LIBERTY, E. Optimal Quantile Approximation in Streams. In: **Symp. on Foundations of Computer Science (FOCS)**. [S.l.: s.n.], 2016. p. 71–78. ISSN 0272-5428.

KAY, M. et al. When (ish) is my bus?: User-centered visualizations of uncertainty in everyday, mobile predictive systems. In: ACM. **Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems**. [S.l.], 2016. p. 5092–5103.

KAZEMITABAR, S. et al. Geospatial stream query processing using Microsoft SQL Server StreamInsight. **Proc. of the VLDB Endowment**, v. 3, n. 1-2, p. 1537–1540, 2010. ISSN 21508097.

KENNEY, J.; KEEPING, E. **Mathematics of Statistics**. Van Nostrand company, 1954. (Mathematics of Statistics, v. 1). Disponível em: <https://tinyurl.com/ybpyupad>.

KICK the bar chart habit. **Nature Methods**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved. SN -, v. 11, p. 113 EP –, 01 2014. Disponível em: <https://doi.org/10.1038/nmeth.2837>.

KINKELDEY, C. et al. Evaluating the effect of visually represented geodata uncertainty on decision-making: systematic review, lessons learned, and recommendations. **Cartography and Geographic Information Science**, Taylor and Francis, v. 44, n. 1, p. 1–21, 2017.

KOSARA, R.; BENDIX, F.; HAUSER, H. Parallel sets: Interactive exploration and visual analysis of categorical data. **IEEE Transactions on Visualization and Computer Graphics**, IEEE Educational Activities Department, v. 12, n. 4, p. 558–568, jul. 2006. ISSN 1077-2626.

KRISHNAMOHAN, K.; FARMWALD, P.; WARE, F. **Prefetching into a cache to minimize main memory access time and cache size in a computer system**. Google Patents, 1996. US Patent 5,499,355. Disponível em: <http://www.google.com/patents/US5499355>.

LANDSEA, C. W. H. R.; FRANKLIN, J. L. **Atlantic Hurricane Database Uncertainty and Presentation of a New Database Format.** 2013. Mon. Wea. Rev., 141, 3576-3592.

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford Large Network Dataset Collection**. 2014. <http://snap.stanford.edu/data>.

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford large network dataset collection**. Mars 2017.

LI, S. et al. Geospatial big data handling theory and methods: A review and research challenges. **{ISPRS} Journal of Photogrammetry and Remote Sensing**, p. –, 2015. ISSN 0924-2716. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0924271615002439>.

Lins, L.; Klosowski, J. T.; Scheidegger, C. Nanocubes for real-time exploration of spatiotemporal datasets. **IEEE Transactions on Visualization and Computer Graphics**, v. 19, n. 12, p. 2456–2465, Dec 2013. ISSN 1077-2626.

LIU, Z.; HEER, J. The effects of interactive latency on exploratory visual analysis. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 2122–2131, Dec 2014. ISSN 1077-2626.

LIU, Z.; HEER, J. The Effects of Interactive Latency on Exploratory Visual Analysis. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 20, n. 12, p. 2122–2131, 2014.

LIU, Z.; JIANG, B.; HEER, J. immens: Real-time visual querying of big data. In: **Proceedings of the 15th Eurographics Conference on Visualization**. Chichester, UK: The Eurographs Association &#38; John Wiley &#38; Sons, Ltd., 2013. (EuroVis '13), p. 421–430. Disponível em: <http://dx.doi.org/10.1111/cgf.12129>.

LOPES, R. H. C. Kolmogorov-smirnov test. In: ____. **International Encyclopedia of Statistical Science**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 718–720. ISBN 978-3-642-04898-2. Disponível em: <https://doi.org/10.1007/978-3-642-04898-2_326>.

MACIEJEWSKI, R. et al. Automated box-cox transformations for improved visual encoding. **IEEE transactions on visualization and computer graphics**, IEEE, v. 19, n. 1, p. 130–140, 2013.

MACIEJEWSKI, R. et al. A visual analytics approach to understanding spatiotemporal hotspots. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 16, n. 2, p. 205–220, 2010.

MAGDY, A.; MOKBEL, M. **Kite**. Mars 2017. <http://kite.cs.umn.edu>.

MAGDY, A. et al. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. **Proc. - International Conference on Data Engineering**, p. 172–183, 2014. ISSN 10844627.

MAGDY, A. et al. Venus: Scalable Real-Time Spatial Queries on Microblogs with Adaptive Load Shedding. **IEEE Transactions on Knowledge and Data Engineering**, v. 28, n. 2, 2016. ISSN 10414347.

MALI, G. et al. A new dynamic graph structure for large-scale transportation networks. In: ____. **Algorithms and Complexity: 8th International Conference, CIAC 2013, Barcelona, Spain, May 22-24, 2013. Proc.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 312–323.

MARCUS, A. et al. **Twitinfo**. New York, New York, USA: ACM Press, 2011. 227 p.

MARTINS, R. M.; MINGHIM, R.; TELEA, A. Explaining neighborhood preservation for multidimensional projections. In: **CGVC**. [S.l.: s.n.], 2015.

MATEJKA, J.; FITZMAURICE, G. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics Through Simulated Annealing. In: ACM. **Proc. Conference on Human Factors in Computing Systems (CHI)**. [S.l.], 2017. p. 1290–1294.

MCGLINN, R. J. A parallel version of cook and kim's algorithm for presorted lists. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 19, n. 10, p. 917–930, set. 1989. ISSN 0038-0644.

Miranda, F. et al. Topkube: A rank-aware data cube for real-time exploration of spatiotemporal data. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 3, p. 1394–1407, March 2018. ISSN 1077-2626.

MonetDB B.V. **GeoSpatial | MonetDB**. <https://tinyurl.com/yal5gwev>. Accessed: 2019-01-19.

MORA, B. Naive ray-tracing: A divide-and-conquer approach. **ACM Trans. Graph.**, ACM, v. 30, n. 5, p. 117:1–117:12, out. 2011. ISSN 0730-0301.

MORITZ, D.; FISHER, D. What Users Don't Expect about Exploratory Data Analysis on Approximate Query Processing Systems. In: ACM. **Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics**. [S.l.], 2017. p. 9.

MORITZ, D. et al. Trust, but Verify: Optimistic Visualizations of Approximate Queries for Exploring Big Data. In: ACM. **Proc. Conference on Human Factors in Computing Systems (CHI)**. [S.l.], 2017. p. 2904–2915.

MORRIS, B.; TRIVEDI, M. Learning trajectory patterns by clustering: Experimental studies and comparative evaluation. In: **2009 IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2009. p. 312–319. ISSN 1063-6919.

MORTON, G. M. A computer oriented geodetic data base and a new technique in file sequencing. International Business Machines Company New York, 1966.

MORTON, K. et al. Support the data enthusiast: Challenges for next-generation data-analysis systems. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 6, p. 453–456, fev. 2014. ISSN 2150-8097.

MUNSHI, A.; SHARMA, V.; SHARMA, S. Lessons learned from cohort studies, and hospital-based studies and their implications in precision medicine. In: **Progress and Challenges in Precision Medicine**. [S.l.]: Elsevier, 2017.

NEPOMNYACHIY, S. et al. What, where, and when: Keyword search with spatio-temporal ranges. In: **Proceedings of the 8th Workshop on Geographic Information Retrieval**. [S.l.]: ACM, 2014. (GIR '14), p. 2:1–2:8. ISBN 978-1-4503-3135-7.

NYC Taxi and Limousine Commission. **Taxi Trip Records**. <https://tinyurl.com/q66cby3>. Accessed: 2018-07-18.

NYPD. **NYC Police Department announces street closures and expected traffic delays for October 11-12th 2014**. <https://tinyurl.com/ybyooj4w>. Accessed: 2018-07-18.

OMIDVAR-TEHRANI, B.; AMER-YAHIA, S.; LAKSHMANAN, L. Cohort representation and exploration. In: IEEE. **DSAA**. [S.l.], 2018.

PAHINS, C.; POZZER, C. Improving divide-and-conquer ray-tracing using a parallel approach. In: **Proceedings of the 2014 27th SIBGRAPI Conference on Graphics, Patterns and Images**. [S.l.]: IEEE Computer Society, 2014. p. 9–16. ISBN 978-1-4799-4260-2.

PAHINS, C. A. L.; COMBA, J. L. D. Similarity-Based Visual Exploration of Very Large Georeferenced Multidimensional Datasets. In: IEEE. **Workshop on Data Systems for Interactive Analysis (DSIA)**. [S.l.], 2016.

Pahins, C. A. L. et al. Hashedcubes: Simple, low memory, real-time visual exploration of big data. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 671–680, Jan 2017. ISSN 1077-2626.

PENG, J. et al. Aqp++: Connecting approximate query processing with aggregate precomputation for interactive analytics. In: ACM. **Proceedings of the 2018 International Conference on Management of Data**. [S.l.], 2018. p. 1477–1492.

PERALTA, R. et al. Similarity-Based Visual Exploration of Very Large Georeferenced Multidimensional Datasets. In: ACM. **Symposium On Applied Computing (SAC)**. [S.l.], 2018.

PETERS, T. **TimSort**. 2002. <http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.

PHILLIPS, J. M. Coresets and sketches. **arXiv preprint arXiv:1601.00617**, 2016.

POLYANOVSKY, V. O.; ROYTBERG, M. A.; TUMANYAN, V. G. Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences. **Algorithms for molecular biology**, BioMed Central, 2011.

PostGIS. **Spatial and Geographic Objects for PostgreSQL**. 2019. <https://tinyurl.com/ycptpbsv>. Accessed: 2019-01-19.

POTTER, K.; GERBER, S.; ANDERSON, E. W. Visualization of Uncertainty without a Mean. **IEEE Computer Graphics and Applications**, v. 33, n. 1, p. 75–79, Jan 2013. ISSN 0272-1716.

POTTER, K. et al. Visualizing Summary Statistics and Uncertainty. In: WILEY ONLINE LIBRARY. **Computer Graphics Forum**. [S.l.], 2010. v. 29, n. 3, p. 823–832.

POTTER, K.; ROSEN, P.; JOHNSON, C. R. From Quantification to Visualization: A Taxonomy of Uncertainty Visualization Approaches. In: DIENSTFREY, A. M.; BOISVERT, R. F. (Ed.). **Proc. IFIP Working Conference on Uncertainty Quantification**. [S.l.]: Springer Berlin Heidelberg, 2012. p. 226–249. ISBN 978-3-642-32677-6.

QUARTERONI, A. The Role of Statistics in the Era of Big Data: A Computational Scientist' Perspective. **Statistics & Probability Letters**, 2018. ISSN 0167-7152.

RAMSAK, F. et al. **Integrating the UB-tree into a database system kernel.** 2000. 263–272 p.

ROSENTHAL, J. S. **A First Look at Rigorous Probability Theory**. Second. [S.l.]: World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2006. xvi+219 p. ISBN 978-981-270-371-2; 981-270-371-3.

ROUSSELET, G. A.; FOXE, J. J.; BOLAM, J. P. A Few Simple Steps to Improve the Description of Group Results in Neuroscience. **European Journal of Neuroscience**, Wiley Online Library, v. 44, n. 9, p. 2647–2651, 2016.

SACHA, D. et al. The Role of Uncertainty, Awareness, and Trust in Visual Analytics. **IEEE Transactions on Visualization and Computer Graphics**, v. 22, n. 1, p. 240–249, Jan 2016. ISSN 1077-2626.

SAMET, H. **Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)**. [S.l.]: Morgan Kaufmann Publishers Inc., 2005. ISBN 0123694469.

SHIEH, J.; KEOGH, E. isax: Indexing and mining terabyte sized time series. In: **Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. [S.l.]: ACM, 2008. p. 623–631. ISBN 978-1-60558-193-4.

SHRIVASTAVA, N. et al. Medians and Beyond: New Aggregation Techniques for Sensor Networks. In: **Proc. International Conference on Embedded Networked Sensor Systems (SenSys)**. [S.l.: s.n.], 2004. p. 239–249.

SIDDIQUI, T. et al. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. **VLDB**, 2016.

SINGH, S.; GUPTA, P. Comparative study id3, cart and c4. 5 decision tree algorithm: a survey. **International Journal of Advanced Information Science and Technology (IJAIST)**, v. 27, n. 27, p. 97–103, 2014.

SISMANIS, Y. et al. Dwarf: Shrinking the petacube. In: **Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data**. [S.l.]: ACM, 2002. (SIGMOD '02), p. 464–475. ISBN 1-58113-497-5.

SpatiaLite. **SpatiaLite**. 2019. <https://tinyurl.com/d9re6ss>. Accessed: 2019-01-19.

STATS, I. L. **Twitter Usage Statistics**. Mars 2017. <http://www.internetlivestats.com/twitter-statistics>.

STOLPER, C. D.; PERER, A.; GOTZ, D. Progressive visual analytics: User-driven visual exploration of in-progress analytics. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 1653–1662, Dec 2014. ISSN 1077-2626.

TOSS, J. et al. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. **Computers and Graphics**, v. 76, p. 117 – 128, 2018. ISSN 0097-8493. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0097849318301390>.

US Bureau of Transportation Statistics. **Air Travel Consumer Report**. <https://tinyurl.com/yde9nwo4>. Accessed: 2018-07-18.

US Department of Transportation. **On-Time Performance Dataset**. <https://tinyurl.com/y7tngze8>. Accessed: 2018-07-18.

VALDIVIA, P. et al. Wavelet-based Visualization of Time-varying Data on Graphs. In: IEEE. **Proc. Conference on Visual Analytics Science and Technology (VAST)**. [S.l.], 2015. p. 1–8.

VARTAK, M. et al. Seedb: efficient data-driven visualization recommendations to support visual analytics. **VLDB**, 2015.

Wang, Z. et al. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 681–690, Jan 2017. ISSN 1077-2626.

WEISSGERBER, T. L. et al. Beyond Bar and Line Graphs: Time for a New Data Presentation Paradigm. **PLoS Biology**, Public Library of Science, v. 13, n. 4, p. e1002128, 2015.

WICKHAM, H. Asa 2009 data expo. **Journal of Computational and Graphical Statistics**, v. 20, n. 2, p. 281—-283, 2011.

WICKHAM, H. **Bin-summarise-smooth: a framework for visualising large data**. [S.l.], 2013.

WICKHAM, H.; STRYJEWSKI, L. 40 Years of Boxplots. **Am. Statistician**, 2011.

WILKINSON, L. Visualizing Big Data Outliers Through Distributed Aggregation. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 256–266, Jan 2018. ISSN 1077-2626.

WITTEN, I. H.; FRANK, E.; HALL, M. A. **Data Mining: Practical Machine Learning Tools and Techniques**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 0123748569, 9780123748560.

WONGSUPHASAWAT, K. et al. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. **TVCG**, IEEE, 2016.

WU, E.; BATTLE, L.; MADDEN, S. R. The case for data visualization management systems: Vision paper. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 10, p. 903–906, jun. 2014. ISSN 2150-8097.

YOON, S.-E.; MANOCHA, D. **Cache-Efficient Layouts of Bounding Volume Hierarchies**. 2006. 507–516 p.

ZAHARIA, M. et al. **Spark: Cluster Computing with Working Sets**. 2010.

ZHANG, H. et al. In-Memory Big Data Management and Processing: A Survey. **IEEE Transactions on Knowledge and Data Engineering**, v. 27, n. 7, p. 1920–1948, 2015. ISSN 10414347.

ZHOU, W. et al. Large scale nearest neighbors search based on neighborhood graph. In: **2013 International Conference on Advanced Cloud and Big Data**. [S.l.: s.n.], 2013. p. 181–186.

Appendices

Appendix A SIMILARITY-BASED VISUAL EXPLORATION OF VERY LARGE GEOREFERENCED MULTIDIMENSIONAL DATASETS

Published in: ACM/SIGAPP Symposium On Applied Computing, 2019.
DOI: 10.1145/3297280.3297556
Qualis CAPES: A1
Authors: Roger Peralta-Aranibar, **Cícero A. L. Pahins**, João L. D. Comba and Erick Gomez-Nieto.

A.1 Abstract

Big data visualization is a main task for data analysis. Due to its complexity in terms of volume and variety, very large datasets are unable to be queried for similarities among entries in traditional Database Management Systems. In this paper, we propose an effective approach for indexing millions of elements with the purpose of performing single and multiple visual similarity queries on multidimensional data associated with geographical locations. Our approach makes use of Z-Curve algorithm to map into 1D space considering similarities between data. We support our proposal by comparisons with state-of-the-art methods in the literature. Additionally, we present a set of results using real data of different sources and we analyze the insights obtained from the interactive exploration.

A.2 Introduction

Along the years, scalability has been one of the main concerns for Database Management Systems (DBMS). This requirement has been addressed in order to support the increasing data volume produced by emerging technologies for data collection as GPS in mobile devices or Internet of Things (IoT). Due to its simplicity for usage and popularity, most of the massive applications – e.g. social networks, e-market or geographical information systems – index a very large number of entries into a DBMS where Relational (as PostgreSQL, MySQL), NoSQL (as MongoDB, Cassandra) and Graph-based (as Neo4j, OrientDB) types. However, critical information retrieval operators, as similarity queries. are not supported into these traditional systems, forcing the analyst to employ additional tools or implement his/her own operators for obtaining similar instances.

Querying for similarities into big databases is a challenging task for data analysis, in general, due to the expensive calculation of similarities among a high number of entries. In this context, the complexity of calculation is totally dependent of two crucial features: *distance measure* and *data dimensionality*. The first aspect is essential to be considered since depending on the type of data, a suitable distance measure will be chosen, for instance, Euclidean and Manhattan distances for numerical data, Jaccard distance for categorical data and Gower distance for mixed data. Logically, all of the measures mentioned above differ on their formulation, whereas complex data type, complexity increases. Second, modern applications make use with a very high number of attributes in order to describe accurately objects, impacting dramatically the cost of similarity calculation. These significant drawbacks impair the addition of similarity querying into DBMS context.

However, some strategies have been proposed to mitigate these difficulties. For instance, dimensionality reduction methods allow us to alleviate the distance calculation into high-dimensional space through mapping data into a lower dimensional space preserving as much as possible the original distances. In addition, some recent data structures overcome data volume drawbacks through preprocessing (Miranda et al., 2018; Wang et al., 2017), distributed storage (CARPENTER; HEWITT, 2016) and parallel processing (DORAISWAMY et al., 2016).

In this work, we propose a data structure to support interactive similarity queries in real time over millions of georeferenced multidimensional entries for visualization. It relies on the combination of dimensionality reduction methods and spatiotemporal data structures. Additionally, our method provides a visual exploration in real-time of similar data by querying one or more entries simultaneously into a geographic map. This main feature opens a wide range of applications for analysis tasks as the search for similar events, behaviors and even reactions in social networks on different geographic locations.

The main contributions of this work can be summarized as follows:

- A new data structure based on dimensionality reduction to index Big Data by similarity, allowing real-time querying.
- A methodology for visually exploring similarities among massive multidimensional datasets associated with geographic locations.

As far as we known, no other techniques are devoted to performing similarity

Table A.1: Summarized differences between data structures addressed for this research.

Features	**ImMens**	**NanoCubes**	**HashedCubes**	**Gaussian Cubes**	**STIG**	**TOPKUBES**	**SemTree**	**Neighborhood Graph**
Multi-Spacial	✓	✗	✓	✗	✓	✗	✗	✓
Multi-Temporal	✓	✗	✓	✗	✓	✓	✗	✓
Categorical	✓	✓	✓	✓	✗	✓	✗	✗
Incremental	✗	✗	✗	✗	✗	✗	✓	✗
Key Saturation	✗	✓	✓	✓	✗	✓	✗	✗
Paralelism	✓	✗	✗	✗	✓	✗	✓	✗
Repeated *mem (de)alloc*	✗	✗	✓	✗	-	✓	-	-
Database Integration	✗	✗	✓	✗	✗	✗	✓	✓
Truncate Pivots	✗	✗	✓	✗	✗	✗	✗	✗
Source Code	✓	✓	✓	✗	✓	✗	✗	✗

queries on multidimensional data associated with geographic locations in real-time for visual exploration tasks.

A.3 Related Work

Data cubes based structures use aggregations that are performed over a dataset to summarize the data and plot it on a geographic map. An aggregation is a typical operation for extracting features and patterns in addition to provide several types of visualizations such us heatmaps, histograms, bar charts, etc. Data cubes is a data structure adopted by many relational databases, it stores the result of performing every aggregation to avoid repeated calculation each time it is executed. It pre-computes every possible aggregation over the dimensions resulting in a massive memory consumption. Nanocubes (Lins; Klosowski; Scheidegger, 2013) makes use of shared links to reduce the space taken by a Data cubes identifying repeated aggregations across the data structure so that every parent node has the information of its children nodes. Nanocubes uses a quadtree for the spatial dimension, a flat tree for categorial dimensions and a sparse summed-area table, making not possible to use multiple spatial dimensions nor multiple temporal dimensions. Another disadvantage is that Nanocubes does not allow updates to the data.

Hashedcubes (Pahins et al., 2017) as an alternative to Nanocubes is a simpler data structure which uses only a linear type array called *Hash* to store delimiter pivots to containing the partially sorted data on every dimension. An advantage of Hashedcubes is the capability of allowing multiple spatial and temporal dimensions interleaves the construction of each dimension. Also, it requires less memory than a Nanocubes but a worst-case for query speed is slower. All the methods mentioned above provide visualization based on the summaries given by the computed aggregations; however, Gaussian Cubes (Wang et al., 2017) also pre-computes the best multivariate Gaussian distribution for a given set to fit models over the data such as linear least squares and principal components analysis

(PCA). Gaussian Cubes requires an extra space to store the multivariate Gaussian distributions which are to be queried in order to reduce the latency. Another data structure based on Nanocubes is TopKubes (Miranda et al., 2018), which identifies what are the most top-k objects in a dataset encoding a measure in a new special dimension including ranking information.

Even though all these emerged data structures had reduced the memory requirement of the classical data cube, the total size keeps growing exponentially as the number of dimensions increases.

GPU optimized methods improves the performance of a data structure carrying the execution of processes concurrently among several cores. STIG (DORAISWAMY et al., 2016) is a data structure that makes use of parallelism in both CPU and GPU. The first part of the search is made in a KD-tree to split the data into buckets of an equal number of elements so that the use of cores in GPU with an NVIDIA board is optimized. The data in each bucket must be stored contiguously and the Kd-tree must be balanced not allowing new inserts. STIG also has several strategies to query the data structure including the use of hybrid CPU and GPU, GPU-only, and GPU with dynamic parallelism. Due to STIG primarily uses a KD-tree, it only allows to index data which can calculate an average.

Semantic-aware approaches includes semantic information that adds an essential meaning to the data and the relationships that lie between them. As an example we have SemTree (AMATO et al., 2015) that is a distributed version of a KD-tree to handling semantic documents, it stores a representation for each document with a set three attributes: <subject, predicate, object>. SemTree finds a partition where the node is to be inserted and splitting if necessary resulting with two types of partitions, for searching and storing data. Another interesting approach is based on graphs (ZHOU et al., 2013). It divides the dataset into groups, having on each group a pivot to represent the cluster in order to query only the pivots instead of the entire dataset.

Table A.1 summarizes the above-mentioned methods where we can conclude initially that HashedCubes have the most features due to its simplicity and the possibility of representing many attributes spatial and temporal dimensions. Instead of not supporting many temporal dimensions such as the TOPKUBES method, the latter maintains the limitations that NanoCubes presents, it does not have the capacity to support many spatial dimensions and it can not be updated if a record is deleted. Similarly, Gaussian Cubes presents the same characteristics of NanoCubes but allows modeling of the dataset.

A.4 Our approach

We are given a large, spatio-temporal, multidimensional dataset defined by location and some other attributes. Then, we are looking for the elements of a set that is close to a given query element under some similarity criterion. Due to the use of N-dimensional datasets the complexity to compute the distance between elements is proportional to the number of N. Hence, our key idea is to transform the elements from N-dimensional into 1-dimension space using a projection technique in order to take advantage of range search over 1-dimensional methods.

A.4.1 Projecting data to 1D

In this section, we discuss three projecting methods called Fiedler Vector, Z-Order Curve (MORTON, 1966), and Hilbert Curve. Then we compare them using neighborhood preservation (MARTINS; MINGHIM; TELEA, 2015), a well-known technique to measure how well the relationship of the data is kept compare the projected space against the original in high-dimension.

The Fiedler vector is obtained calculating the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix of a graph G which represents the given dataset. The graph is defined by connecting the data to its closest neighbors that lies inside a threshold, and the Laplacian matrix is defined as:

$$L = D - A \tag{A.1}$$

Where D is the degree matrix and A is the adjacency matrix of the graph. The resulting vector is the projected data to 1-dimension.

The other projection method considered is called Z-Order Curve, one of its main advantages over other space-filling curves is the simplicity of its calculation. The z-value of a point in a N-dimensional space is obtained by the simple process of bit-shuffling. In Figure A.1 is shown an example of a 2D Z-curve and bit shuffling process.

Finally, our last method is the Hilbert curve, an alternative to the Z-Order Curve in which the main advantage is to avoid long jumps when describing the curve. In Figure A.2 we show the Hilbert curve map between 1 and 2 dimension that reasonably well preserves locality.

Figure A.1: Z-curve algorithm explanation.

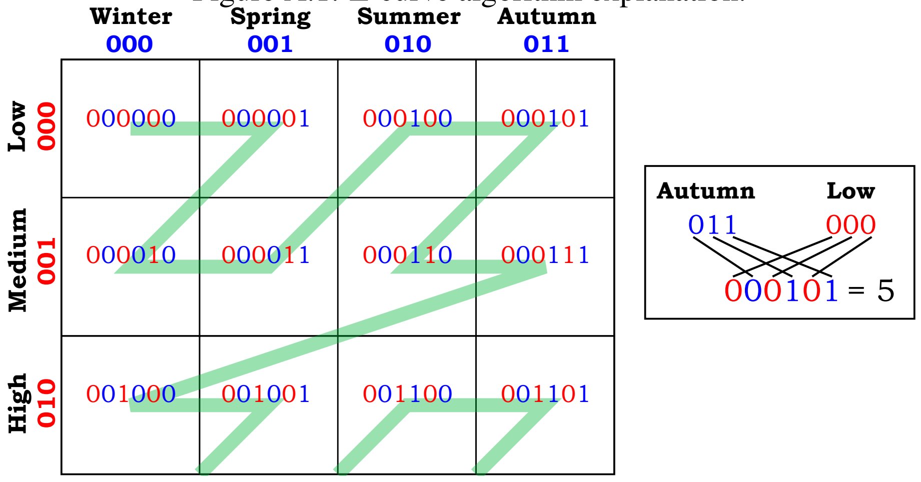
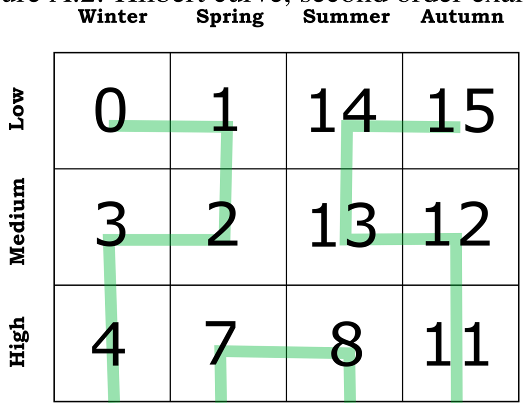


Figure A.2: Hilbert curve, second order example.



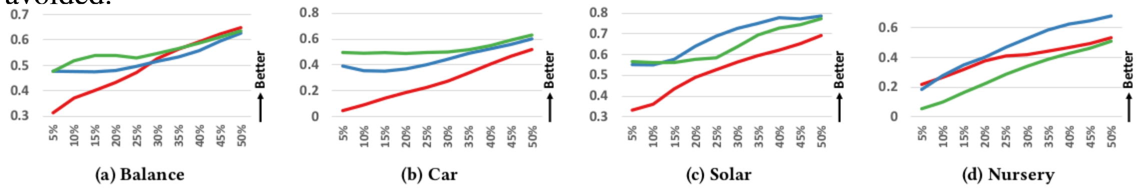
To determine which of the previously described methods works best when preserving the locality, we collected four datasets to be tested by our metric measure neighborhood preservation. Each dataset contains an array of numbers ranged in size up to 13000 entries and domain-specific values with up to six distinct values. We summarize all of the schema variations and datasets in Table A.2.

The balance dataset contains 625 total entries with four dimensions, each of these having five different values. Every possible point is instanced and appears only once in the dataset, so they are fully described by both projection curves; Z-order and Hilbert. In this case, the Hilbert curve projection has the best accuracy due to all the jumps that could have occurred in the dataset are minimized (Figure A.4.1).

Table A.2: Summary of the datasets used to evaluate Fiedler vector, Z-order and Hilbert curve projections.

dataset	objects	dimensions
balance	625	4
car	1728	6
solar	1389	10
nursery	12960	8

Figure A.3: Quantifying the neighborhood preservation of four datasets performed by Fiedler Vector (🟥), Z Order Curve (🟦) and Hilbert Curve (🟩), the result of this measure is impacted by the instances distribution along the dimension. The best method to use depends in how well the data is described by a Z curve or how many times a long jump is avoided.



A similar case takes place in the car dataset that contains 1728 entries with a square-like structure having six dimensions of a range between three and four distinct values. As we can see in Figure A.4.1 the best accuracy is also obtained by Hilbert Curve projection because of the almost full square-like structure in the dataset mentioned before. Nevertheless, our next solar dataset has a more irregular definition because it holds a total of 1389 entries, ten dimensions, and an unbalanced range of values from two to six variations. We can observe in Figure A.4.1 that the Z-order curve acquire the best performance, as a result of not having the long jumps that the Hilbert curve is outstanding.

In our last example with the nursery dataset (Figure A.4.1), the Z-order keeps better the relationship of the dataset, this is explained by the fact that the dataset of 12960 of 8 dimensions is better described by a Z-curve since its possible values are ranged from two to five possible values.

Now, we will explain how to build and query our method. We will explain why we use specific techniques and why they work well together as shown in Figure A.4. First, we deal with the complexity of working in a high dimensional space even after having to make queries in this reduced space improving the memory and speed latency.

A.4.2 Indexing data

At this stage, data must be projected onto a 1-dimensional. To accomplish this task we could use either the Z-Order or Hilbert Curve projection. Because of its discrete values, the aggregation function can be easily used to put closer similar values together. With this new data representation in 1-dimension, we can take the concept of the Hashed-Cubes and use it to index in a single hashed array due to its simplicity and benefits over the others methods.

Figure A.4: A summary of our proposed methodology for similarity queries in a high dimensional space. The main idea is to pre-process the high dimensional dataset to obtain a simplified but descriptive dataset in one-dimensional space to enable the computation of similarity queries in real-time

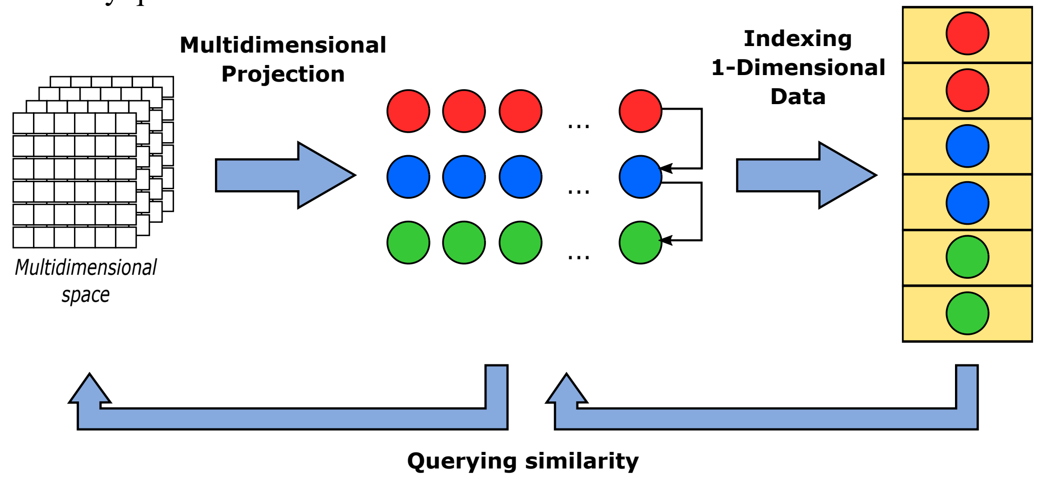
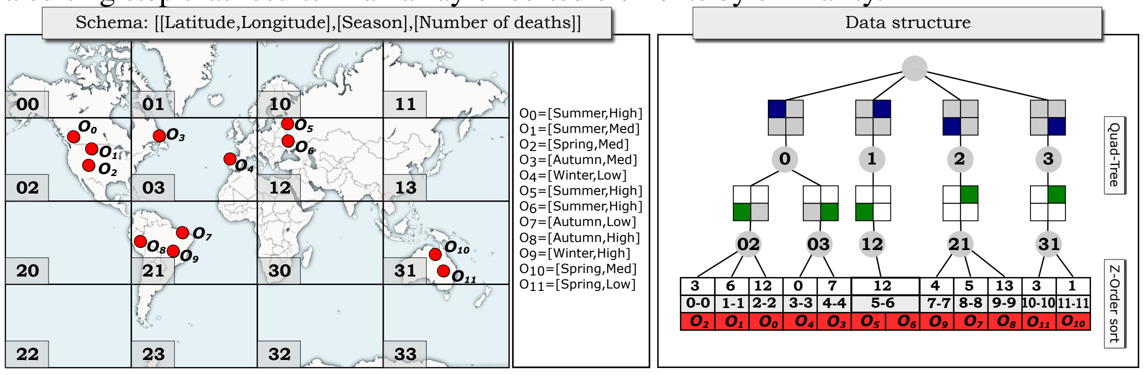


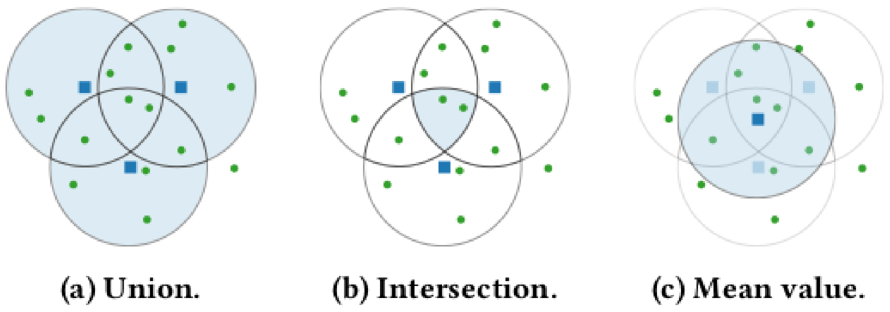
Figure A.5: An overview of our method. First we obtain a single value representation given by a multidimensional projection for every record in the dataset, so that we execute a sorting step that results in an array of sorted elements by similarity.



We are interested to know where the most similar elements are placed on a geographical map given a group of points, following the idea of HashedCubes we sorted the data by the spatial attributes. We keep using the Quadtree data structure, therefore for each note, we obtain a pair of pivots delimiting the data. The goal is to split the location of the most similar elements given a threshold value. For that reason we now sort the data inside each pivot using its similarity value, that is, the projection into 1D space described above. We illustrate how we build our data structure in Figure A.5.

It is important to note that besides we only show how to index by geographical position and similarity, our method also supports categorical, and temporal attributes as its predecessor.

Figure A.6: Concrete example of the our approaches with the same three query points showing different results in every case.



A.4.3 Querying data

As HashedCubes approach we traverse from root until the area we want to plot. Then, we obtain only the pivots which values are under some range criteria. The most natural similarity query for a specific area happens when a single point is queried. However, in the multiple points case, we propose three approaches: Union, Intersection and Mean value.

The first and straightforward way is to return all the items that are close to the query points. To carry out this task we iterate for all the pivots that contain the most similar data and then plot them following its spatial attributes.

The second strategy is to retrieve the common points that are inside the range query of all the query points since the HashedCubes structure already sorts the indexed values this approach does not impact drastically in the performance of our data structure.

Our last approach generates an artificial point by computing the mean value for all attributes of the entire set of query points. Once this point is processed, we perform a range query using our threshold parameter and retrieving all points contained.

We illustrate the already explained approaches to retrieve data in Figure A.6

A.4.4 Implementation details

The client-server architecture is also used in our implementation. After pre-computing the multidimensional projection, this new particular 1-dimensional attribute is store along with the other data to be read by an event loop. The server is a C++ implementation to exploit the operation of a pre-allocated chunk of memory to avoid and to re-use existing data structures provided by STL among other libraries.

We have built a front-end visualization to query the data that is inserted using Javascript, HTML5, SVG, and D3.js. Consider Figure A.5 that visualizes the similarity distribution of Brightkite datasets, which consist of 4.5 millions of check-ins that range from April 2008 to October 2010, on a geographical map. We use a total of five heatmaps with different palette colors where each heatmap covers progressively the area containing most similar elements.

In order to not disrupt the fluent interaction, we added three widgets in which the user is able to update the similarity parameters that are wanted to be shown by our method, we describe them as follows:

Points selection

adds an unlimited number of points with many selection boxes depending on the number of attributes. These points are sent to the server to perform the query operation. The user can also change the multi-query operation dynamically.

Range query input

takes five integer numbers sorted from least to highest. These five elements represent the range query values shown on the map with a heatmap of the color, next to the entered range query value. We also include a checkbox to enable the density of data for each heatmap plotted.

Similarity histogram

are located at the bottom of the interface. This widget perceptively displays the similarity distribution of our explored dataset. Additionally, we show a highlight bar with the purpose of indicating where the point queries lay down along with a blue area. It represents the amount of data covered that will be shown on the geographical map.

A.5 Results

In this section, we evaluate our method against three publicity-available datasets. In addition, we show the difference between using the three of our methods to merge the multi-query operation. In Table A.3, we summarize the relevant information for project-

Table A.3: Summary of the datasets considered for our results and the resource usage when indexing the categorical values for similarity queries.

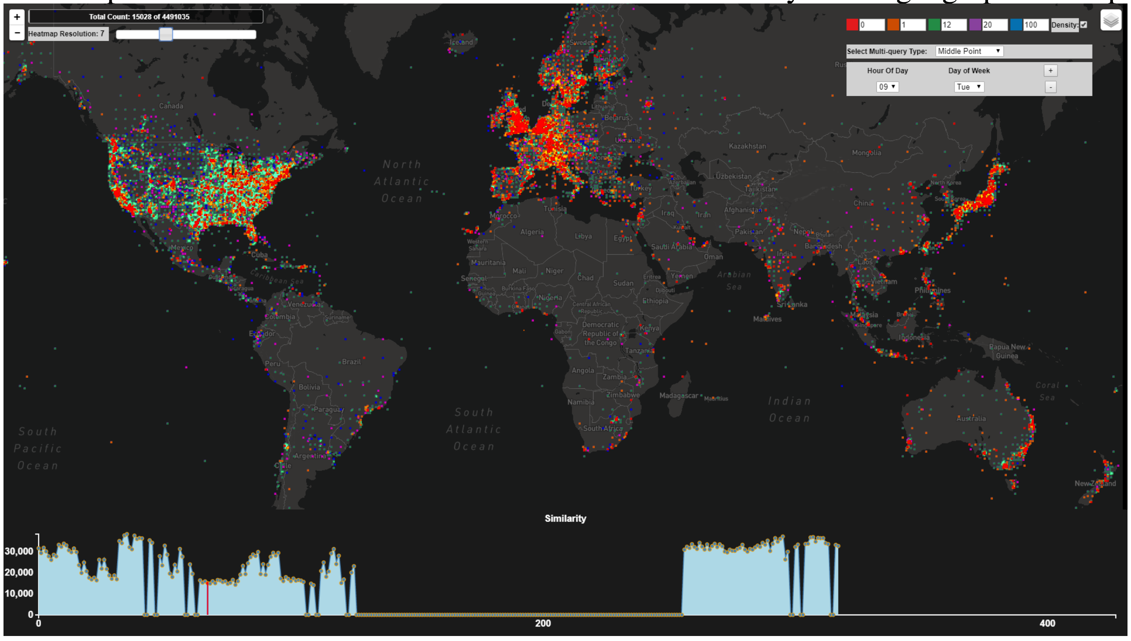
dataset	entries(E)	Z-order	hilbert	pivots	schema
brightkite	$4.5M$	317	299	168	lat, lon, hour of the day(24), day of month(31)
NYPD incident	415901	427	508	197	lat, lon, jurisdiction for incident(18), classification code(16)
twitter	$210.6M$	751	695	131	lat, lon, app(4), device(5), language(15)

ing and indexing the dataset on the previously described datasets. The total number of records E, the maximum value that a Z-order, and Hilbert curve projection would have, are expressed in the next two columns. The following column represents the number of “pivots” created when indexing the projected values, and finally a short description of the schema of each dataset.

Our first result is using a set of four million users check-ins on Brightkite (CHO; MYERS; LESKOVEC, 2011b) dataset. The raw data consists of five dimensions: User, Date, Time, Latitude and Longitude. In Figure A.7 we can visualize the similarity distribution of indexing the Date and Time over the location (i.e. Lat and Lon). Using the point query of hour of day (value of 9) and day of week (value of Tuesday) the most similarity elements are on the sides of United States (East and West sides), most of data are located on Europe, specifically in United Kingdom, Netherlands, and Germany (from a color palette from red to blue as shown at the top-right corner). Additionally, we can observe a uniform distribution of the data in the similarity histogram at the bottom of the interface with two well-separated clusters. As a result, the most of the heatmap colors range are present on the map.

Our next dataset also has four million entries of New York City Civilian Complaints. The input data, which was made available as csv files, has 415901 entries. In this example, we show four different query points (Figure A.8) in which we can see the cluster distribution of similarity depending on where the point query lies down as we can see in the histogram above each image moreover the similarity clusters are more separated resulting in interesting color patterns. For example, given the color palette at the top-right corner of each image, the query point is right in the mayor similarity cluster of the dataset.

Figure A.7: Exploring the Brightkite checking similarity distribution using five different colormaps that enables a visual differentiation of the similarity over a geographical map.


Total Count: 15028 of 4491035

Heatmap Resolution: 7

🟥 0 🟧 1 🟩 12 🟪 20 🟦 100 Density: ✓

Select Multi-query Type: Middle Point

Hour Of Day: 09

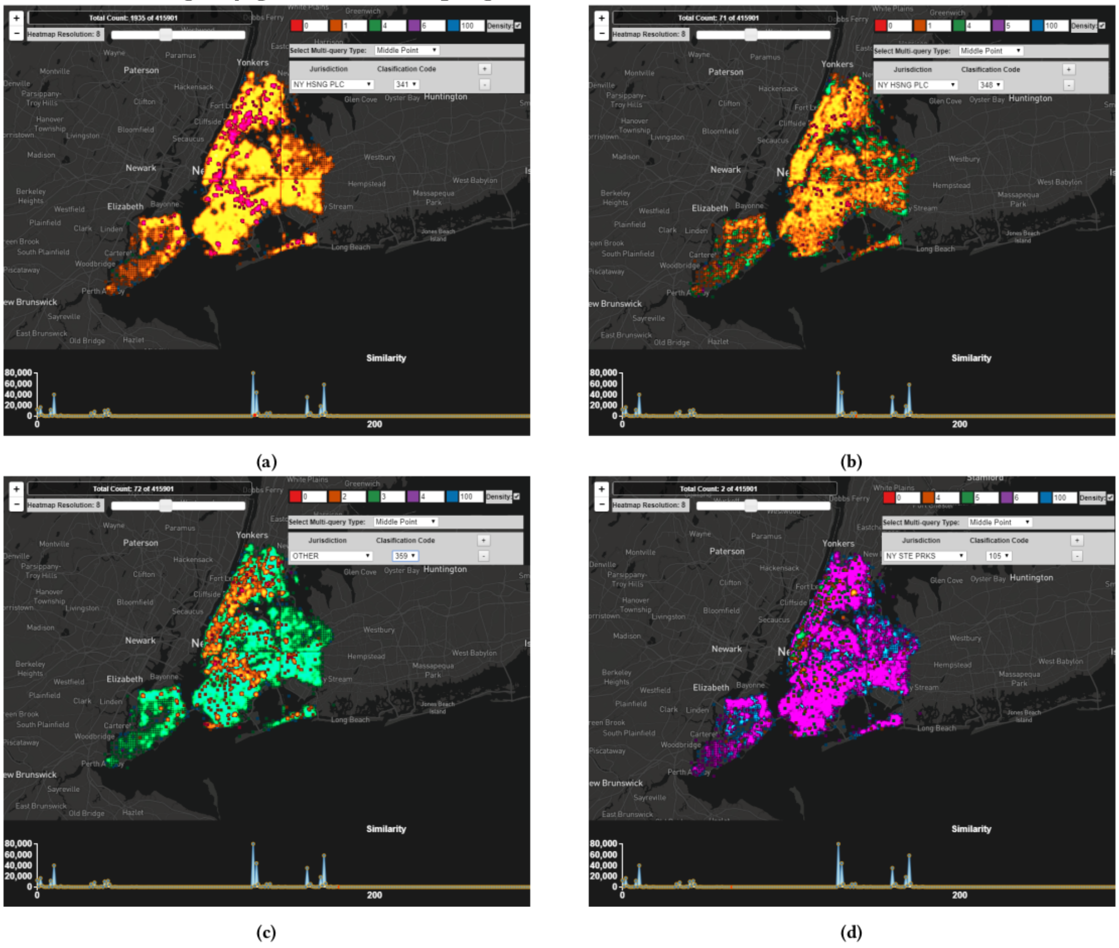
Day of Week: Tue

We can see this in the similarity histogram at the bottom of the interface hence the color of the plotted heatmap is mostly the closest ranges colors meaning the most similar elements (Figure A.5). In the next case of Figure A.5 the query point is in the middle of two similarity groups, that is why the green of the color palette has appeared at the around the corners. The same criteria happen in Figure A.5 because the point is now further from a cluster, so the green part of the heatmap appears in more regions. However, in our last example (Figure A.5) the heatmap of the less similar elements is more relevant since the point query is more isolated.

Our last dataset has tweets collected from the United States during the dates of November of 2011 to June of 2012. The total amount of geolocated tweets is about 210 million and each tweet consists of a spatial information; device, application, and language used as categorical information, respectively 5, 4 and 15 distinct values.

In this example, we use the index to show the difference in applying our three implemented methods, i.e. Union, Intersection and Mean value (Figure A.6. In Figure A.9, there are three different color maps with different retrieval methods but with the same three query points. The first query point refers to a “None” value for Language, Device, and Application. The next query point has a language value of “Italian”, “android” for the device, and “Instagram” value for the application. The final query point has a “russian” value for language, the device is an “iPad” and send through the “twitter” application. The Union plot example has a majority of red color showing that all the points are inside

Figure A.8: Visual exploration of the New York City Civilian Complaints dataset using four different query points and range queries.



(a) Total Count: 1935 of 415901; Heatmap Resolution: 8; Select Multi-query Type: Middle Point; Jurisdiction: NY HSNG PLC; Clasification Code: 341

(b) Total Count: 71 of 415901; Heatmap Resolution: 8; Select Multi-query Type: Middle Point; Jurisdiction: NY HSNG PLC; Clasification Code: 348

(c) Total Count: 72 of 415901; Heatmap Resolution: 8; Select Multi-query Type: Middle Point; Jurisdiction: OTHER; Clasification Code: 359

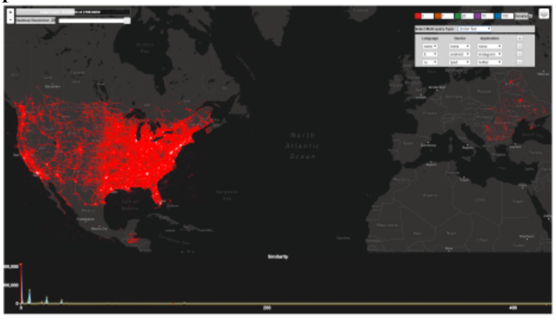
(d) Total Count: 2 of 415901; Heatmap Resolution: 8; Select Multi-query Type: Middle Point; Jurisdiction: NY STE PRKS; Clasification Code: 105

the union set of the three query points. However, in the case of intersection, we appreciate a purple dominance since all the attributes are far from the specific intersect set. Using a mean point to query against, we have two sections showing that all the points are closer to the obtained query point and that in the East part of United States are the closest values.

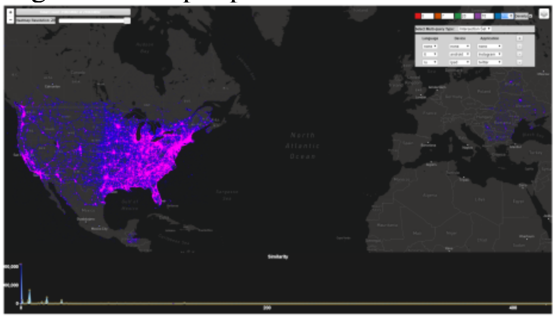
A.6 Discussion and Limitation

To the best of our knowledge, none of the methods in the state-of-the-art in exploratory model visualization offers a similarity visualization over a large, multidimensional, spatiotemporal datasets. Our proposed method is able to plot a similarity distribution of a dataset giving a set of querying points over a geographical map as seen in Section A.5. We also have demonstrated its value showing three different cases with an evaluation of our retrieval techniques with public datasets. However, there are still many

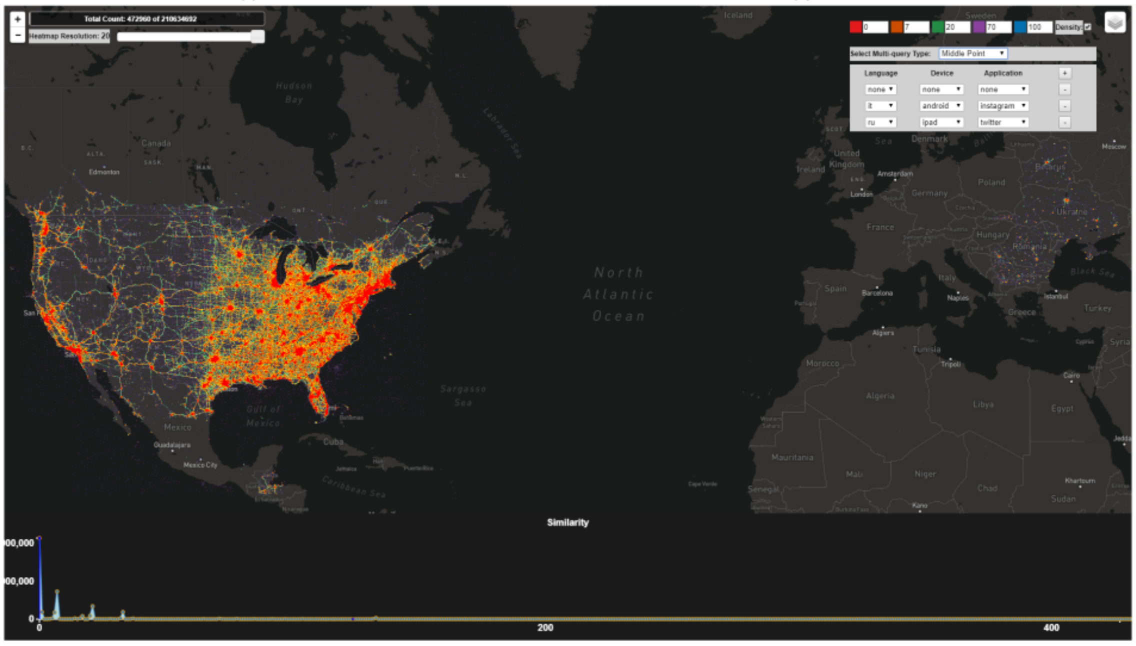
Figure A.9: Visualizing 210 million public geolocated Twitter showing our three approaches to retrieve the most similar elements against multiple points.



(a) Union.



(b) Intersection.



Total Count: 472960 of 210634692

Heatmap Resolution: 20

🟥 0 🟧 7 🟩 20 🟪 70 🟦 100 Density: ✓

Select Multi-query Type: Middle Point

Language	Device	Application
none	none	none
it	android	instagram
ru	ipad	twitter

(c) Mean value.

limitations and opportunities for improvement.

Projecting data to a one-dimensional space reduces the complexity of dealing with a high dimensional dataset. Additionally, we only make use of the Z-order and Hilbert curve projections to project categorical attributes dismissing other dimension types, such as temporal or spatial attributes. Even though we can use categorical attributes it is only possible to the ones with ordinal nature due to we need to establish the distance measure between any pair of elements.

Furthermore, obtaining the resulting projected value for each element in the dataset results in an integer value ranging from zero to a number that is proportional to the number of categorical attributes and the possible values of each of this attributes. For instance, if the explored dataset contains many categorical attributes and each one of these attributes can take a high number of possible values, the resulting projection will spread the entire projected dataset. Also, it will increase the number of pivots generated by a HashedCubes, impacting negatively in both memory usage and running time.

Finally, the choice between the Z-order and Hilbert curve to be used by our method depends on the overall distribution of values of our dataset. If all the instances of the dataset are well described inside a square described precisely by a Z description then the Z-order curve is our best option; otherwise, as longer jumps occur in continuous instances of the dataset the more accuracy the Hilbert Curve projection will have. This selection relies on the analyst preference to decide which projecting method to use.

A.7 Conclusion

In this paper, we present a novel data structure that combines HashedCubes and dimensionality reduction methods in order to provide a set of similarity queries to perform on very large datasets. Resulting of the above-mentioned combination, our method also shares its features. Such as scalability with a high number of dimensions, performance in terms of querying time, and the impossibility of updating the data structure once is built. However, we include an essential feature to increase the range of queries to be performed.

Currently, to determine the color transfer function of each similarity heatmaps we provide a panel that shows the color palette that is used for each query, and a color scale values that are hidden to the user. With the purpose of providing a more perceptive interaction, we believe that a widget to set this values dynamically would impact positively on our implemented framework.

Appendix B PACKED-MEMORY QUADTREE: A CACHE-OBLIVIOUS DATA STRUCTURE FOR VISUAL EXPLORATION OF STREAMING SPATIOTEMPORAL BIG DATA

Published in: Elsevier Computers & Graphics Journal (CG), 2018.
DOI: 10.1016/j.cag.2018.09.005
Qualis CAPES: A2
Authors: Júlio Toss, **Cícero A. L. Pahins**, Bruno Raffin and João L. D. Comba.

B.1 Abstract

The visual analysis of large multidimensional spatiotemporal datasets poses challenging questions regarding storage requirements and query performance. Several data structures have recently been proposed to address these problems that rely on indexes that pre-compute different aggregations from a known-a-priori dataset. Consider now the problem of handling *streaming* datasets, in which data arrive as one or more continuous data streams. Such datasets introduce challenges to the data structure, which now has to support dynamic updates (insertions/deletions) and rebalancing operations to perform self-reorganizations. In this work, we present the Packed-Memory Quadtree (PMQ), a novel data structure designed to support visual exploration of streaming spatiotemporal datasets. PMQ is *cache-oblivious* to perform well under different cache configurations. We store streaming data in an internal index that keeps a spatiotemporal ordering over the data following a quadtree representation, with support for real-time insertions and deletions. We validate our data structure under different dynamic scenarios and compare to competing strategies. We demonstrate how PMQ could be used to answer different types of visual spatiotemporal range queries of streaming datasets.

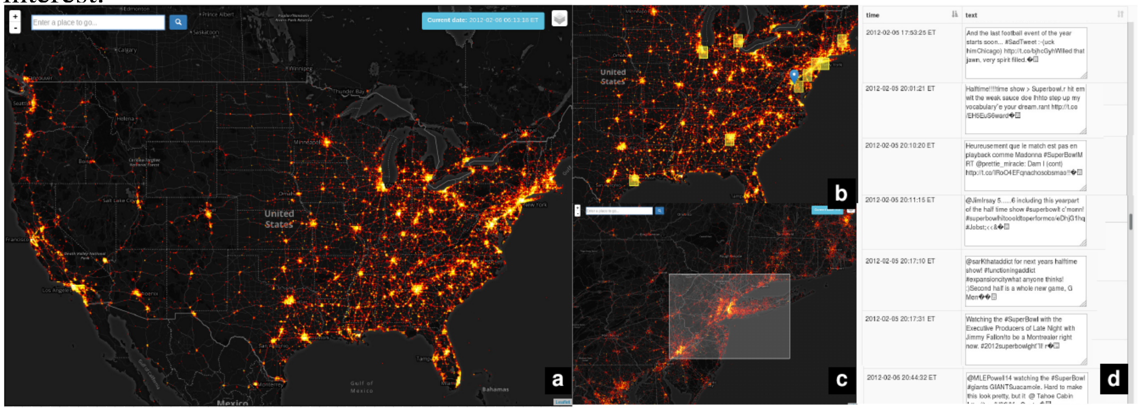
B.2 Introduction

Advanced visualization tools are essential for big data analysis. Most approaches focus on large static datasets, but there is a growing interest in analyzing and visualizing data streams upon generation. Twitter is a typical example. The stream of tweets is continuous, and users want to be aware of the latest trends. This need is expected to grow with the Internet of things (IoT) and massive deployment of sensors that generate large and heterogeneous data streams. Over the past years, several in-memory big-data management systems have appeared in academia and industry. In-memory databases systems avoid the overheads related to traditional I/O disk-based systems and have made possible to perform interactive data-analysis over large amounts of data. A vast literature of systems and research strategies deals with different aspects, such as the limited storage size and a multi-level memory-hierarchy of caches (ZHANG et al., 2015). Maintaining the right data layout that favors locality of accesses is a determinant factor for the performance of in-memory processing systems.

Stream processing engines like Spark (ZAHARIA et al., 2010) or Flink (APACHE. . . , 2017) support the concept of *window*, which collects the latest events without a specific data organization. It is possible to trigger the analysis upon the occurrence of a given criterion (time, volume, specific event occurrence). After a window is updated, the system shifts the processing to the next batch of events. There is a need to go one step further to keep a live window continuously updated while having a fine grain data replacement policy to control the memory footprint. The challenge is the design of dynamic data structures to absorb high rate data streams, stash away the oldest data to stay in the allowed memory budget while enabling fast queries executions to update visual representations. A possible solution is the extension of database structures like R-trees (GUTTMAN, 1984) used in SpatiaLite (SpatiaLite, 2019) or PostGis (PostGIS, 2019), or to develop dedicated frameworks like Kite (MAGDY; MOKBEL, Mars 2017) based on a pyramid structure (MAGDY et al., 2014; MAGDY et al., 2016).

In this paper, we propose a novel self-organized cache-oblivious data structure, called Packed-Memory Quadtree (PMQ), for in-memory storage and indexing of fixed length records tagged with a spatiotemporal index. We store the data in an array with a controlled density of gaps (*i.e.*, empty slots) that benefits from the properties of the *Packed Memory Arrays* (BENDER; DEMAINE; FARACH-COLTON, 2005). The empty slots guarantee that insertions can be performed with a low amortized number of data movements ($O(\log^2(N))$) while enabling efficient spatiotemporal queries. During insertions, we rebalance parts of the array when required to respect density constraints, and the oldest data is stashed away when reaching the memory budget. To spatially subdivide the data, we sort the records according to their Morton index (GARGANTINI, 1982), thus ensuring spatial locality in the array while defining an implicit, recursive quadtree, which leads to efficient spatiotemporal queries. We validate PMQ for consuming a stream of

Figure B.1: A Twitter stream is consumed in real-time, indexed and stored in the Packed-Memory Quadtree. **(a)** : live heat-map displays tweets in the current time window. **(b)** : alerts indicate regions with high activity of Twitter posts at the moment. **(c)** : the interface allows to drill-down into any region and query the current data. **(d)** : the actual data can be retrieved from the Packed-Memory Quadtree to analyze the tweets in the region of interest.



time	text
2012-02-05 17:53:25 ET	And the last football event of the year starts soon... #SadTweet :-(uck himChicago) http://t.co/bjhcGyhWlled that jawn, very spirit filled.�
2012-02-05 20:01:21 ET	Halftime!!!!time show > Superbowl.r hit em wit the weak sauce doe lhhto step up my vocabulary'e your dream.rant http://t.co/EH5EuS6ward�
2012-02-05 20:10:20 ET	Heureusement que le match est pas en playback comme Madonna #SuperBowlM RT @prettie_miracle: Dam I (cont) http://t.co/lRoO4EFqnachosobsmao!!�
2012-02-05 20:11:15 ET	@JimIrsay 5......6 including this yearpart of the half time show #superbowlt c'monn! #superbowlhitoooldtoperformco/eDhjG1hq #Jobst;<<&�
2012-02-05 20:17:10 ET	@sarKthataddict for next years halftime show! #functioningaddict #expansioncitywhat anyone thinks! :)Second half is a whole new game, G Men��
2012-02-05 20:17:31 ET	Watching the #SuperBowl with the Executive Producers of Late Night with Jimmy Fallon!to be a Montrealer right now. #2012superbowlght'!! r�
2012-02-05 20:44:32 ET	@MLEPowell14 watching the #SuperBowl #giants GIANTSuacamole. Hard to make this look pretty, but it @ Tahoe Cabin

tweets to answer visual and range queries. Figure B.1 shows the user interface prototype built to support the data analysis process. PMQ significantly outperforms the widely adopted spatial indexing data structure R-tree, typically used by relational databases, as well as the conjunction of Geohash and B$^+$-tree, typically used by NoSQL databases (FOX et al., 2013). In summary, we contribute (1) a self-organized cache-oblivious data structure for storing and indexing large streaming spatiotemporal datasets; (2) algorithms to support *real-time* visual and range queries over streaming data; (3) performance comparison against tried and trusted indexing data structures used by relational and non-relational databases.

B.3 Related Work

In building an efficient system to enable interactive exploration of data streams, we must deal with challenges common to areas like in-memory big-data, stream processing, geospatial processing, and information visualization.

Data Structures.. Data structures need to dynamically process streams of geospatial data while enabling the fast execution of spatiotemporal queries, such as the *top-k* query that ranks and returns only the k most relevant data matching predefined spatiotemporal criteria. One approach is to store data continuously in a dense array following the order given by a space-filling curve, which leads to desirable data locality. Inserting an element takes on average $O(n)$ data movements, i.e., the number of elements to move to make room for the newly inserted element. The cost of memory allocations can be reduced using an amortized scheme that doubles the size of the array every time it gets full. However, elements are often inserted in batches in an already sorted array. In that case, one approach is to use adaptive sorting algorithms to take advantage of already sorted sequences (ESTIVILL-CASTRO; WOOD, 1992; COOK; KIM, 1980; MCGLINN, 1989). Timsort (PETERS, 2002) is an example of an adaptive sorting algorithm with efficient implementations. We show experiments that compare our data structure to Timsort. Another possibility is to rely on trees of linked arrays. The B-tree (BAYER; MCCREIGHT, 1972) and its variations (BRODAL; FAGERBERG, 2003) are probably the most common data structure for databases. The UB-Tree is a B-tree for multidimensional data using space-filling curves (RAMSAK et al., 2000). These structures are seldom used for in-memory storage with a high insertion rate. They are competitive when data access time is large enough compared to management overheads, often the case for on-disk storage. Such data structures are *cache-aware*, *i.e.*, to ensure cache efficiency they require a calibration according to the cache parameters of the target architecture.

Sparse arrays are an alternative that lies in between dense arrays and trees of linked arrays. Data is stored in an array larger than the actual number of elements to store, using the extra room to make insertions and deletions more efficient. Itai et al. (ITAI; KONHEIM; RODEH, 1981) were probably the first to propose such data structure. Bender et al. (BENDER; DEMAINE; FARACH-COLTON, 2005; BENDER; HU, 2007a) refined it, leading to the Packed Memory Array (PMA). The main idea is that by maintaining a controlled spread of gaps, insertions of new elements can be performed moving much fewer than $O(N)$ elements. The insertion of an element in the PMA only requires $O(\log^2(N))$ amortized element moves. This cost goes down to $O(\log(N))$ for random insertion patterns. Bender and Hu (BENDER; HU, 2007a) also proposed a more complex PMA, called adaptive PMA, that keeps this $O(\log(N))$ for specific insertion patterns like bulk insertions. PMA is a *cache-oblivious* data structure (FRIGO et al., 1999), *i.e.* it is cache efficient without explicitly knowing the cache parameters. Such data structures are interesting today since the memory hierarchy is getting deeper and more complex with different block sizes. Cache-oblivious data structures are seamlessly efficient in this context. Bender et al. (BENDER; DEMAINE; FARACH-COLTON, 2005; BENDER et al., 2007) also proposed to store a B-tree on a PMA using a van Emde Boas layout, leading to a cache-oblivious B-tree. However, it leads to a complex data structure without a known practical implementation. Still, PMA has few known applications. Mali et al. (MALI et al., 2013) used PMA for dynamics graphs. Durand et al. (DURAND; RAFFIN; FAURE, 2012) relied on PMA to search for neighbors in particle-based numerical simulations. They indexed particles in PMA based on the Morton index computed from their 3D coordinates. They proposed an efficient scheme for batch insertion of elements, while Bender relied on single element insertions. In this paper, we propose to extend PMA for in-memory storage of streamed geospatial data.

Stream Processing and Datacubes Structures.. Stream processing engines, like GeoInsight for MS SQL StreamInsight (KAZEMITABAR et al., 2010), are tailored for single-pass processing of the incoming data without the need to keep in memory a large window of events that require an advanced data structure. The emergence of geospatial databases led to the development of a specialized tree, called R-Tree (GUTTMAN, 1984), that associates a bounding box to each tree node. Several data processing and management tools have been extended to store geospatial data relying on R-trees or variations like the SpatiaLite (SpatiaLite, 2019) extension for SQLite or PostGis (PostGIS, 2019) for PostgreSQL. Our experiments include comparisons with both. Though such spatial libraries brought flexibility for applications in the context of traditional spatial databases, their algorithms are not adapted to consume a continuous data stream. Magdy et al. (MAGDY et al., 2014; MAGDY et al., 2016) proposed an in-memory data structure to query and update real-time streams of tweets. Initially called Mercury, then Venus and eventually Kite (MAGDY; MOKBEL, Mars 2017) for the latest implementation (Kite is also benchmarked in our experiments). They rely on a pyramid structure that decomposes the space into H levels. Periodically the pyramid is traversed to remove the oldest tweets to keep the memory footprint below a given budget. This idea to rely on bounding volume hierarchies is also popular in computer graphics for indexing 3D objects and accelerating collision detection (YOON; MANOCHA, 2006). One difficulty in these data structures is to ensure fast insertions while keeping the tree balanced. The data structure may also become too fragmented in memory leading to an increase of cache misses. The partitioning criteria are based on heuristics. There is often no theoretical performance guarantees.

Finally, we point out that several data structures were proposed recently for the visual analysis of big data. A common theme is the idea of pre-computing aggregations in *datacubes* proposed by Gray et al. (GRAY et al., 1997b). Representative work include imMens (LIU; JIANG; HEER, 2013), Nanocubes (Lins; Klosowski; Scheidegger, 2013), Hashedcubes (Pahins et al., 2017) and Gaussian Cubes (Wang et al., 2017), all designed for processing static data. Streamcube (FENG et al., 2015) combines an explicit spatio-temporal quadtree with datacubes for on-pass hashtag clustering. The PMQ proposes a dynamic data structure supporting the main visual queries described in these works.

B.4 Packed-Memory Quadtree

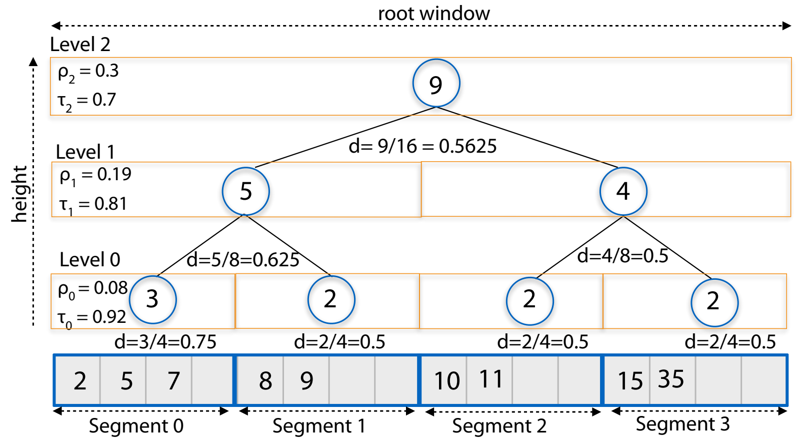
In this section, we explain the PMQ internal organization, update methods, and query types to support stream data analysis. In our description, we used as motivation dataset a stream of tweets, each with spatial location and associated satellite data. Our PMQ builds upon a PMA data structure but departs from the original one on different aspects:

- Data are indexed and sorted according to their Morton index to enforce data locality for efficient spatial queries;
- Data insertions are performed by batches in a top-down scheme to factor rebalance operations, while the PMA inserts elements one by one using a bottom-up scheme;
- The PMQ has a limited memory budget. Once we reach the maximum size and density, we stash the oldest data through a customized process;
- Support for answering geospatial visual queries to allow interactive analysis of streaming datasets.

B.4.1 The PMQ Data Structure

We present here the PMQ data structure that strongly relies on the Packed-Memory Array (ITAI; KONHEIM; RODEH, 1981; BENDER; HU, 2007a). A PMQ is an array with extra space to maintain a given density of (empty) gaps between the (valid) elements. An array of size N (counting the gaps) is divided into $O(N/\log(N))$ consecutive *segments* of size $O(\log(N))$. For convenience, the number of segments is chosen to be a power of 2. PMQ is stored in memory and keeps, for each element, an indexing *key* and associated *value*. Segments are paired hierarchically by *windows* following a tree structure. A level 0 window corresponds to a single segment, while the h level window encompasses the full array. The *density* of a window is the ratio between the number of (valid) elements in

Figure B.2: PMQ with 4 segments: ρ_l and τ_l are the minimum and maximum densities allowed at each level l of PMQ, d the actual window density. The numbers in circles count the valid element per window and are stored in the PMQ accounting array.



the window and its size. As we will see, the PMQ goal is to control the window densities to ensure insertion or removal of elements can be performed at low cost.

The minimum and maximum density bounds for a window at level l are respectively ρ_l and τ_l. We define density bounds such that:

$$\rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0. \tag{B.1}$$

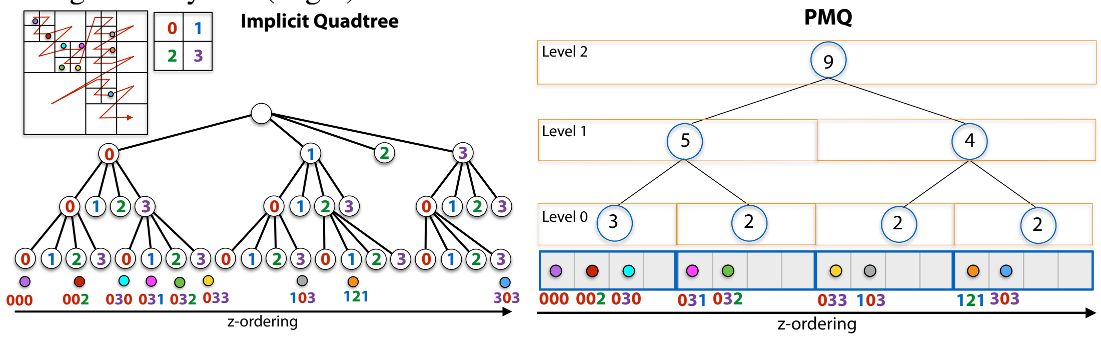
Thus, the larger a window, the more constraining its density bounds. The minimum and maximum densities of windows at intermediate levels are linearly interpolated between the $[\rho_0, \rho_h]$ and $[\tau_h, \tau_0]$ thresholds as defined below:

$$\tau_l = \tau_h + (\tau_0 - \tau_h)\frac{(h-l)}{h}, \tag{B.2}$$

$$\rho_l = \rho_h - (\rho_0 - \rho_h)\frac{(h-l)}{h}, \tag{B.3}$$

and $2\rho_h < \tau_h$. The upper (resp. lower) density threshold decreases (resp. increases) by $O(1/\log(N))$. This $O(1/\log(N))$ interval is fundamental to guarantee that an insertion or deletion requires $O(\log^2(N))$ amortized data movements. A *valid* PMQ is the one that satisfies density values for all windows. To compute a window density in constant time without having to scan its content, we associate an auxiliary *accounting array* to store the number of valid elements per window. This array requires an extra $O(2\frac{N}{\log(N)})$ of memory space. Figure B.2 illustrates a PMQ for 9 elements. The density thresholds are: $\rho_0 = 0.08$, $\rho_2 = 0.3$, $\tau_2 = 0.7$, $\tau_0 = 0.92$, and the values for ρ_1 and τ_1 are defined according to Equation B.2 and Equation B.3.

Figure B.3: **PMQ** storage of 9 z-indexed elements from a 2D domain (right). Z-indices correspond to a quadtree actually never built as z-index are directly computed by interleaving their x-y bits (Right).



B.4.2 Data Indexing

Space-filling curves define a map of multidimensional points to one dimension, which allows to order the data in a 1D array. The PMQ relies on the Morton space-filling curve to store the elements sorted according to their Morton index. The Morton curve enables to linearly index data with 2D coordinates through a low cost bit-level operation, while preserving well the data spatial locality. Data that are close in 2D tend to have close Morton index (also called Z-index or geohash) and thus are stored nearby in the PMQ. Elements with the same Morton index are sorted according to their timestamp (e.g., tweet timestamp).

The Morton curve actually defines a recursive Z-shaped space partitioning that follows a quadtree subdivision. The ordering generated by the Morton curve is equivalent to the ordering produced by a depth-first traversal in a quadtree (BERN; EPPSTEIN; TENG, 1993). The number of bits used for the Z-index defines this quadtree max depth. For the PMQ, this number of bits is static. For each new incoming element, its Z-index is computed from its (x, y) position coordinates by interleaving the bits of x and y, defined according to Equation B.4 and Equation B.5 (note that both equations output integer values). Truncating by 2 bits a Z-index provides the index of the parent cell in the quadtree. Figure B.3 illustrates a PMQ for 9 elements in a 2D domain. Elements are sorted based on their Z-index that recursively defines an implicit quadtree subdivision (never stored).

B.4.3 Dynamic Updates

The PMQ is designed to store a stream of data inserted by batches. The insertion starts at the topmost window of the PMQ (the full array) by checking if a violation of the density bounds occurs after inserting a batch of incoming elements. Consider a valid PMQ filled with K ordered elements. Suppose we want to insert a batch of I new elements stored in an *insertion array*. The goal is to insert these new elements while keeping the PMQ valid. The insertion algorithm follows a top-down approach.

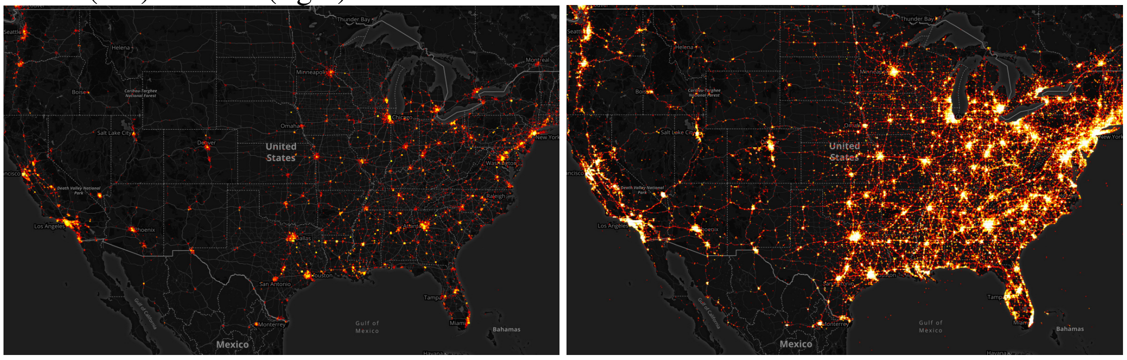
If the density of the full PMQ array goes beyond τ_h counting the I new elements, we first scan the array to count the number of elements with a timestamp older than a given threshold λ. If removing these elements the PMQ meets its density bounds, we remove them, rebalance evenly the remaining elements while inserting the I new elements (sorted first). Otherwise, we perform the same operation but first doubling the PMQ array size. The constraint $2\rho_h < \tau_h$ guarantees that the density of this double size PMQ is above ρ_h.

We now describe the batch insertion. Consider the case where inserting elements does not cause the full array density to go over τ_h. Let p be the key of the first element of the right top window. We re-order the insertion array such that elements smaller than p are on the left of the insertion array, while the others are on the right. The left elements will go in the top left window of PMQ, the others in the top right window. We test for both top windows their new densities against the corresponding thresholds, counting the elements to insert. If at least one top window does not respect the density thresholds, we *rebalance* the elements of the full array while including the new ones, *i.e.*, we evenly redistribute all elements. After the rebalance we have the guarantee that all windows down to segments satisfy their density bounds since densities are less constraining as the window size decreases. Otherwise, density thresholds are respected and the algorithm proceeds recursively. In the best case, *rebalances* only span individual segments. We update the accounting array after each batch insertion.

Note that when performing a rebalance we keep the elements sorted based on their Morton index and insertion timestamp. Since the sorting during rebalancing is stable, the only requirement is that we order the elements in the batch array by arrival time (which is the natural order in a real-time stream). Rebalancing is automatically triggered when needed. No heuristic is needed to decide when to split or delete a node as in (MAGDY et al., 2014; MAGDY et al., 2016). Memory allocations are only needed when doubling the array. We control memory consumption by setting the threshold timestamp λ based on the arrival rate of the data stream. In practice, PMQ self-stabilizes: it doubles its size until reaching a *steady* state where insertions and deletions balance themselves. The accounting array is updated with each window rebalance.

Notice that none of these operation use the lower densities. But they are kept in

Figure B.4: The heatmap is updated dynamically as the stream of tweets is received. With an average insertion rate of $1000\ tweets/sec$ we show the heatmap when PMQ contains $100K$ (left) or $10M$ (right) elements.



the PMQ description and supported in our implementation for completeness. They can be useful for scenarios not evaluated here. They enable to trigger window rebalances when removing elements.

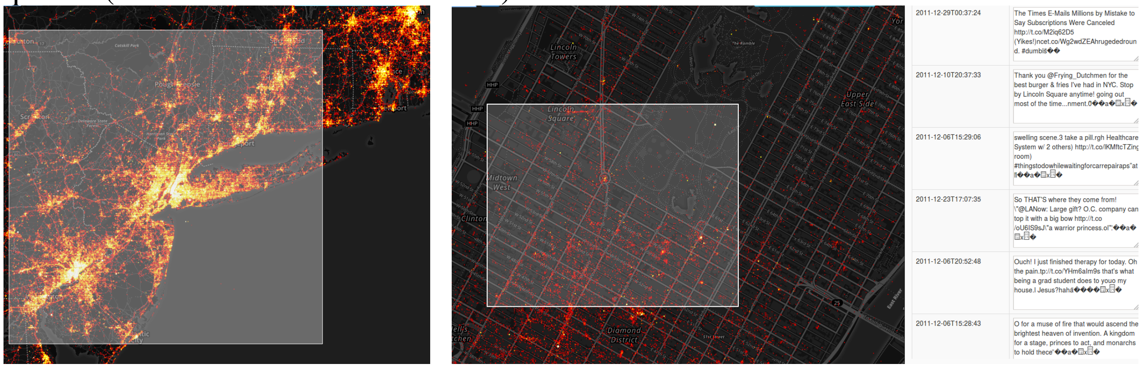
The PMQ is a cache-oblivious data structure as it does not depend on cache parameters. The worst-case amortized cost is $O(\log^2(N))$ per insertion. The proof is given in B.9.

B.4.4 Query Types

We present three types of queries that we support in the current implementation of PMQ: heatmap, range, and top-k queries. Other types of queries can be incorporated if needed.

Heatmap Queries.. The visual interface of our system uses a world heatmap continuously updated based on the content stored in the PMQ (see Figure B.4). An important observation is that Z-cells do not align with the tiles of the heatmap. Also, the interface allows zooming into specific regions of the world, thus needing to map the tiles of the heatmap grid to Z-cells. We compute the *zoom* level ζ in the quadtree of Z-cells corresponding to each heatmap tile. If $\zeta = 0$, we need to aggregate the full PMQ data into a single tile, corresponding to the full PMQ data. Heatmap construction for a single tile consists of counting the number of data samples for each pixel drawn inside the tile. For instance, a tile of $256x256$ pixels corresponding to a quadtree node at level ζ is computed by counting for each pixel the number of elements stored in the corresponding descendant Z-cells at depth $\zeta + 8$. As we only need to *count* the elements per tile (element values are not necessary), we accelerate counting using the accounting array.

Figure B.5: Heatmap zoom and range queries are used to explore the latest stream of tweets around the New York city area. The in-memory storage of PMQ provides fast access to the actual tweets' content allowing real-time user interaction even on large range queries (R = radius of the selected area).



2011-12-29T00:37:24	The Times E-Mails Millions by Mistake to Say Subscriptions Were Canceled http://t.co/M2iq62D5 (Yikes!)ncet.co/Wg2wdZEAhrugededround. #dumbl���
2011-12-10T20:37:33	Thank you @Frying_Dutchmen for the best burger & fries I've had in NYC. Stop by Lincoln Square anytime! going out most of the time...nment.0��a��x��
2011-12-06T15:29:06	swelling scene.3 take a pill.rgh Healthcare System w/ 2 others) http://t.co/lKMftcTZing room) #thingstodowhilewaitingforcarrepairaps"at li��a��x��
2011-12-23T17:07:35	So THAT'S where they come from! \"@LANow: Large gift? O.C. company can top it with a big bow http://t.co /oU6IS9sJ\"a warrior princess.ol"'.��a��x��
2011-12-06T20:52:48	Ouch! I just finished therapy for today. Oh the pain.tp://t.co/YHm6alm9s that's what being a grad student does to youo my house.I Jesus?hahã�����x��
2011-12-06T15:28:43	O for a muse of fire that would ascend the brightest heaven of invention. A kingdom for a stage, princes to act, and monarchs to hold thece"��a��x��

Range Queries.. A range query is a spatial query that requests all elements stored in a rectangular region (Figure B.5). We define a range query by the corners of a bounding box in the map. Given a range query, we have to access the PMQ to retrieve all records within the rectangular region. We return the result to the application for any post-processing of this information. In our interface, we currently display a subset of the results (e.g., a fixed number of tweets). Unlike heatmaps, which queries the PMQ using a fixed resolution grid, the range query can define an arbitrary region. Therefore, we need to find the coarsest Z-cells that contain the bounding box of the range query. Since we do not store the quadtree explicitly, we follow the Z-ordering recursively to find the Z-cells that fully enclose the bounding box. We refine each Z-cell to locate the leaf Z-cells intersecting or included in the bounding box. We refer to the book by Samet (SAMET, 2005) for the range query algorithm for quadtrees. Using the Z-cell indices, we locate through binary searches in the PMQ the ranges that contain the needed elements.

Top-k queries.. The top-k query combines the temporal ordering with the spatial dimension to find the most relevant data according to a given spatiotemporal interval. This query is processed like the range query but filters the candidate values in a temporary priority queue of size k. Given a 2D point p, the top-k query finds the elements with k lowest values according to a score function. The search space of the top-k queries can be reduced using both spatial and time thresholds. The parameter R defines a radius where records are going to be ranked by distance to p. Similarly, the parameter T limits the oldest timestamp to consider in the scoring function. Both scores are then normalized and combined into a final score to balance the importance of the spatial and temporal dimensions. Elements in the same Z-cell (*i.e.*, with the same Morton code) are ordered based on their timestamp. The top-k search uses the same refinement algorithm as for range queries to find the records included inside the bounding box of radius R centered at p. We insert the returned elements into a priority queue of max-size k ordered by the spatiotemporal score to keep only the k elements with the highest score.

B.5 Implementation

We implemented PMQ in C++. Each element has a 64-bit *key* representing the spatial index plus a *value* for storing additional information. We rely on 50-bit length Morton code for the spatial index, defining a quadtree with a fixed depth of 25 levels of refinement. The Morton index is computed from the (x, y) coordinated obtained through the EPSG:3857 projection of the longitude (lon) and latitude (lat) associated with each element, defined as:

$$x = \left(\frac{lon + 180}{360} .2^z \right), \tag{B.4}$$

$$y = \left(1 - \frac{\ln\left(\tan\left(lat.\frac{\pi}{180} \right) + \frac{1}{\cos\left(lat.\frac{\pi}{180} \right)} \right)}{\pi} \right) .2^{z-1}, \tag{B.5}$$

where both $(x, y \in \mathbb{Z} \mid 0 \leq x, y < 4^z)$ and Z is the maximum quadtree depth.

The choice of record content to consider depends on the application needs. If only the metadata is required, we used a 16-byte struct to store latitude (32-bit float), longitude (32-bit float) and insertion timestamp (64-bit unsigned integer).

Our implementation uses PMQ segments of a fixed size of eight elements as we found no significant performance benefit in increasing the segment size with the array size. When rebalancing a window, we always pack data at the left of each segment to favor low-level optimizations like prefetching. Besides giving the window densities, the accounting array is also used to get the position of the last valid element of each segment, to avoid scanning the empty slots.

We target streaming scenarios where the PMQ is used to store the most recent incoming data sorted in memory, keeping as much data as possible for a given memory budget. Data are removed only when the PMQ is full, keeping the PMQ density high enough to avoid reducing its size (waste of memory and time as leading to a PMQ oscillating through cyclic size halving and doubling). Thus rebalances are only triggered when the window upper density bounds are reached. In our experiment we use $\tau_0 = 0.92$ and $\tau_h = 0.7$ that give the best performance tradeoff (see Section B.7.3).

B.6 Examples of Visualization Analysis using PMQ

We present an example of the interactive exploration of tweets enabled by PMQ and its user interface. The dataset consists of geolocated tweets collected with the Twitter API between November 2011 and June 2012 over the United States. The dataset has a total of 210.6 million tweets. We simulate an incoming stream of tweets by grouping them into batches of fixed size and iteratively inserting them into a given data structure.

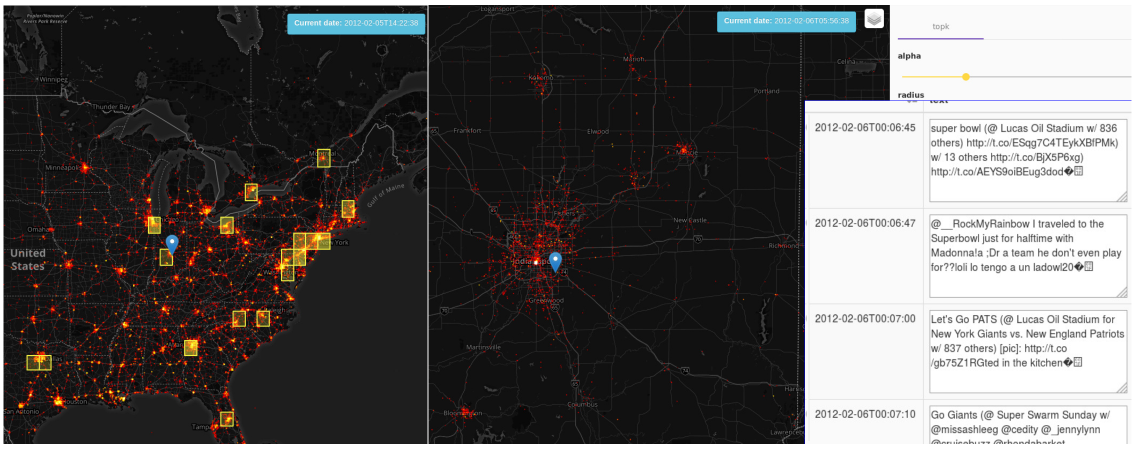
B.6.1 Drill-down exploration of a Twitter stream

We provide a visual interface that allows a drill-down exploration of a tweet stream and support different queries. PMQ supports the storage in memory of the latest tweets, thus filling the gap between stream processing engines working only on a small window of the input stream, and classical solutions on persistent storage. The heatmap enables to display the concentration of tweets posted over the last hours. PMQ can index one second of tweets and keep all the elements sorted in less than 1.5 ms on average, with a maximum at about 1s when a batch triggers a top rebalance with element removal (Table B.3). It is more than capable of keeping up with the stream rate and support visual queries. The interface allows the user to zoom into the heatmap or to perform range queries. We display the tweets inside selected areas in a separate area next to the map. Figure B.5 shows the combined use of heatmaps and range queries at different zoom levels over New York. The user can interactively zoom until finding the desired information.

B.6.2 Allert detection of regions with high tweet rates

Systems like TwitInfo (MARCUS et al., 2011) provide an interface for visualizing real-time Twitter data. Based on the user-given keyword search, the system fetches the matching twitter stream and generates an aggregated higher-level visualization, which is kept up-to-date with the incoming tweets. Such exploratory framework enables a better understanding of on-going events. However, since the stream is filtered with a fixed input keyword, it is not suitable when monitoring unexpected events. One example of an unexpected event is the monitoring of the volume of tweets in a given region. We implemented a simple pre-processing of batches to trigger alerts in regions with high tweet activity. Once an alert is triggered, the user can further investigate it by interactively exploring the last received tweets. During exploration, one can also perform top-k queries to retrieve the top-most relevant tweets at a given point. For instance, on February 5th of 2012 at 14:22 UTC, the system indicates a high tweet activity over Indianapolis. Zooming into the alert zone and using range queries, we observe that many people are at the Lucas Oil Stadium commenting about the Super Bowl game. We set a top-k query at the stadium to follow the most relevant tweets nearby. The filtered feed displayed on the right panel of the interface shows tweets with information like the teams playing (New York Giants Vs. New England Patriots), or about the Madonna's show during half-time (Figure B.6).

Figure B.6: Allert detection: triggers configured by the user show alerts (yellow squares) on several cities with a high rate of tweet arrival during Super Bowl 2012. We select the Indianapolis region (where the game occurs) and filter tweets using a top-k query to retrieve the most relevant tweets in the area.


Current date: 2012-02-05T14:22:38
Current date: 2012-02-06T05:56:38
topk
alpha
radius
text

2012-02-06T00:06:45	super bowl (@ Lucas Oil Stadium w/ 836 others) http://t.co/ESqg7C4TEykXBfPMk) w/ 13 others http://t.co/BjX5P6xg) http://t.co/AEYS9oiBEug3dod�
2012-02-06T00:06:47	@__RockMyRainbow I traveled to the Superbowl just for halftime with Madonna!a ;Dr a team he don't even play for??loli lo tengo a un ladowl20�
2012-02-06T00:07:00	Let's Go PATS (@ Lucas Oil Stadium for New York Giants vs. New England Patriots w/ 837 others) [pic]: http://t.co /gb75Z1RGted in the kitchen�
2012-02-06T00:07:10	Go Giants (@ Super Swarm Sunday w/ @missashleeg @cedity @_jennylynn

B.7 Performance Evaluation

We created a series of benchmarks to evaluate the performance of PMQ. When possible, we showed comparisons against competing solutions from the industry and other open-source libraries. The B-Tree compared in our experiments is the `stx::btree`, an efficient open source implementation of in-memory B+Tree (BINGMANN, 2013). The R-Tree implementation is from the `Boost` C++ template library (BOOST, 2017). We conducted the experiments to allow reproducing results and exploring different parameter configurations. We created a GitHub repository to store supplemental material and additional benchmarks[1]. The benchmarks were run on a dedicated Linux machine with an Intel i7-4790 CPU @ 3.60GHz with 32GB of main memory.

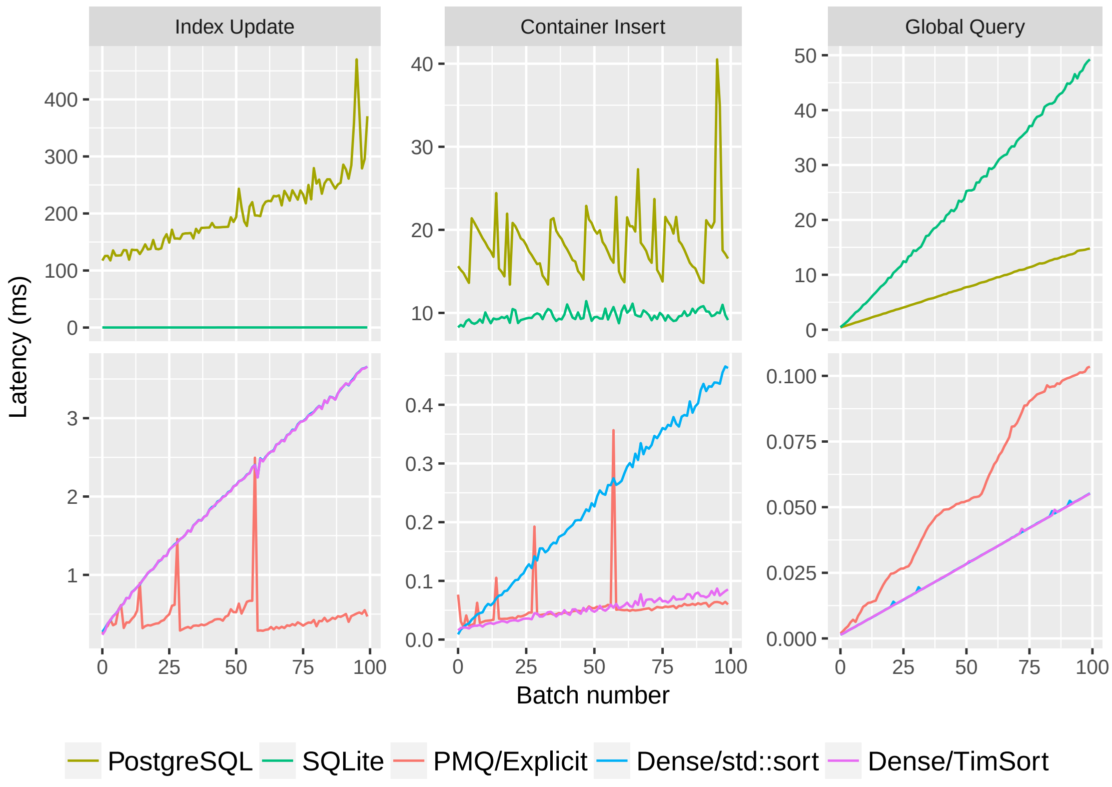
B.7.1 Evaluating Storage Solutions for Spatial Data

We conducted a set of experiments to evaluate PMQ against two different storage solutions for spatial data. The first type of storage solution is traditional open source relational databases (RDBMS) with geospatial library extensions, such as: (1) in-memory SQLite + SpatiaLite and (2) PostgreSQL + PostGIS. SQLite uses in-memory storage, while PostgreSQL uses a disk. Typically, RDBMS store entries in a table and build an additional spatial index separately. This optional index supports efficient spatial queries that contain geometric predicates. The second storage solution are dense vectors using a pointer-based quadtree index on a dense C++ `std::vector`. Spatial ordering in the container uses two sorting algorithms: (1) the C++ `std::sort()` implementation from `GNU GCC libstdc++` and (2) the `C++ TimSort` (GORO, 2016) adaptive sorting algorithm.

This set of benchmarks gives an insight into the scalability of insertion and query operations of the solutions above. The different data structures are initially empty and increase their size as elements arrive in batches. We measured the insertion time of each batch, including the time for updating the index (in case of RDBMS) and physically storing the data. We also measured the time for accessing data from the storage. After each batch insertion, we queried all elements indexed by the data structure (Figure B.7). As can be seen, even with a small number of elements, the database solutions have poor scalability. While PostGIS uses disk storage, it spends most of the time optimizing the index and physically reordering elements on disk. SpatiaLite, on the other hand, seems to have a less efficient indexing strategy than PostGIS. It spends less time on indexing and insertion operations, but pays a significant cost to access the data, even if stored in memory. None of the database solutions are suited to the real-time latency requirements of update and read operations. The spikes on PMQ benchmarks correspond to doubling

[1]<https://github.com/pmq-authors/pmq-extras>

Figure B.7: Performance comparison of spatial data storage solutions. Top row: standard geospatial databases can not handle real-time insertions. Bottom row: in-memory containers based on dense or sparse (PMQ) vectors.

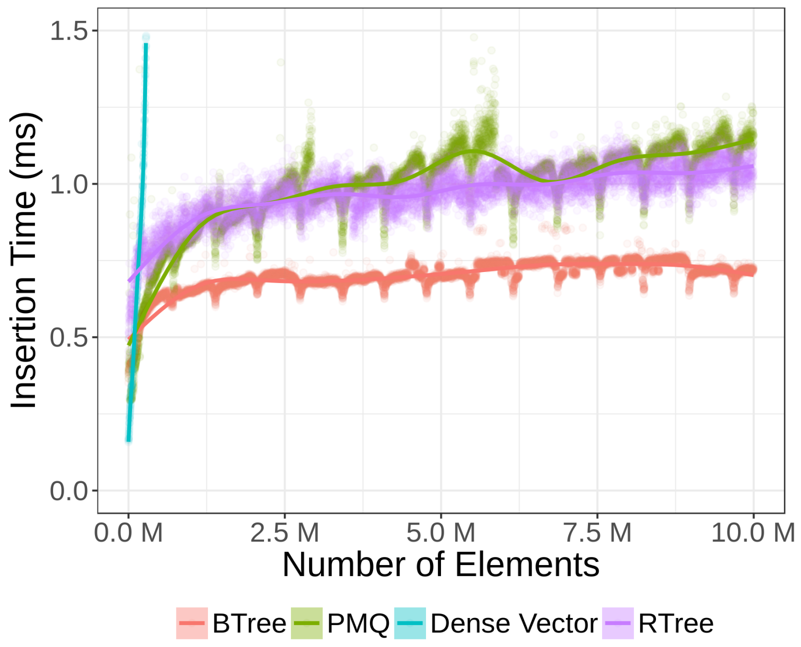


the array size when the structure reaches the maximum density. The sparse storage of PMQ allows reducing the time on insertion when compared to the dense vectors. In the next experiments, we remove the database solutions from the comparisons since they are an order of magnitude slower than the vector-based storage approaches, and compare against low-level structures such as B-trees and R-trees.

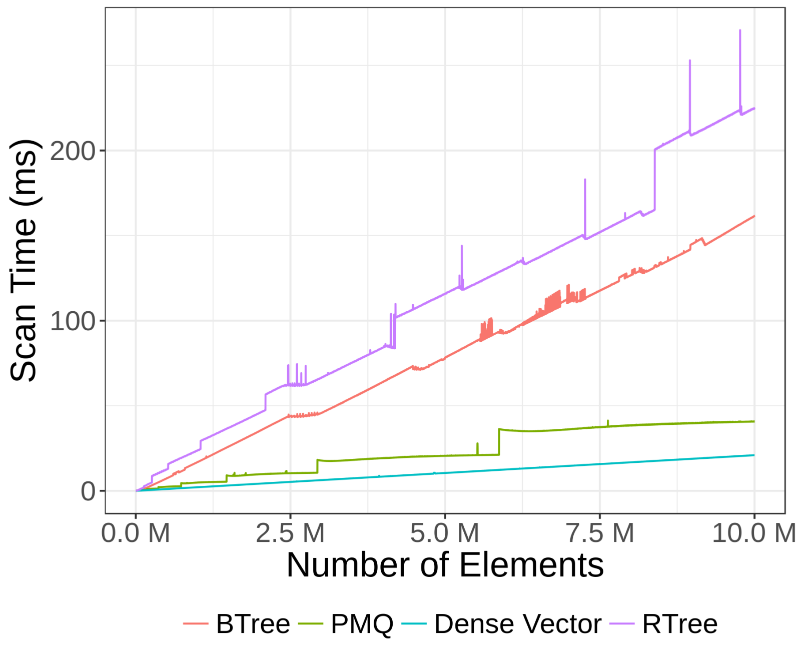
B.7.2 Evaluating Insertions

We evaluate the scalability of the data structures by comparing their trade-offs between insertions and scanning operations. These two operations represent a performance compromise of two conflicting workloads. While tree-based data structures, like B-Tree and R-Tree, show good insertion performance (B.8(a)), they fail in maintaining in-memory data locality, which has a significant impact on scanning operations (B.8(b)). The solution using dense sorted vectors reveals the importance of data locality when scanning. We derive the lower-bounds of scanning performance when we achieve the best locality.

Figure B.8: Scalability insertion and scan operation.



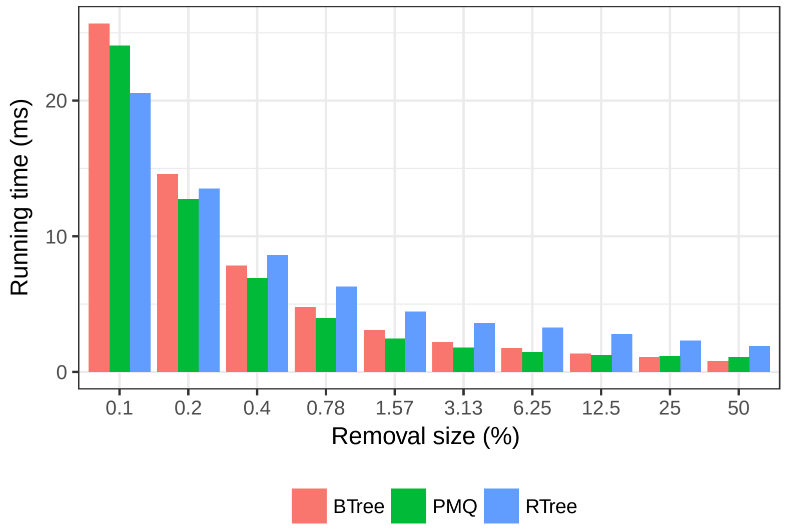
(a) Insertion of $10M$ elements by batches of 1000 elements.



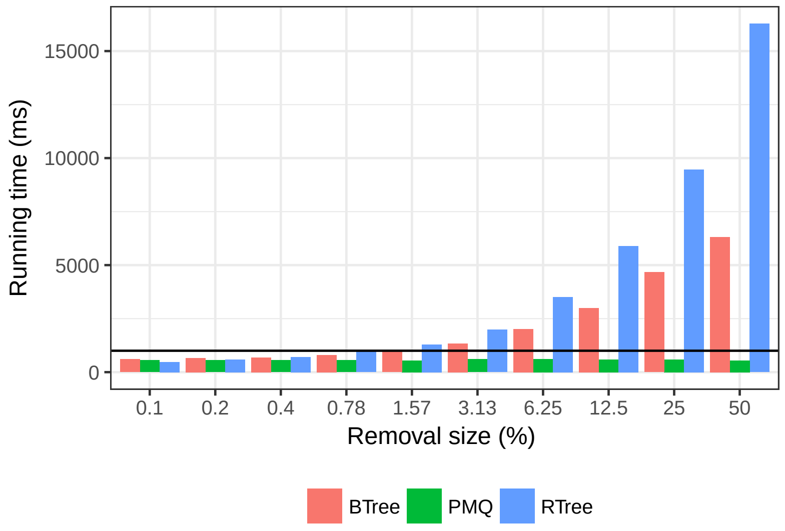
(b) Time for a full scan of the dataset after each batch insertions.

However, its update costs for frequent insertions make it impracticable for large amounts of data. PMQ shows a good compromise between these two operations. On insertions, it performs similarly to the R-Tree, it is 2X times slower than the B-Tree and scales logarithmically with the size of the data structure. At the same time, PMQ pays only a small constant overhead relative to best possible scanning data-structure, the dense vector. Compared to the tree-based data-structures, with 10 M elements, the scan operations on

Figure B.9: Steady data regime: deletions are performed periodically. For each test, we insert a dataset of 46 million elements. The maximum number of elements allowed is half of the dataset size (around 23 million elements), and removals are configured with different percentages of the maximum.

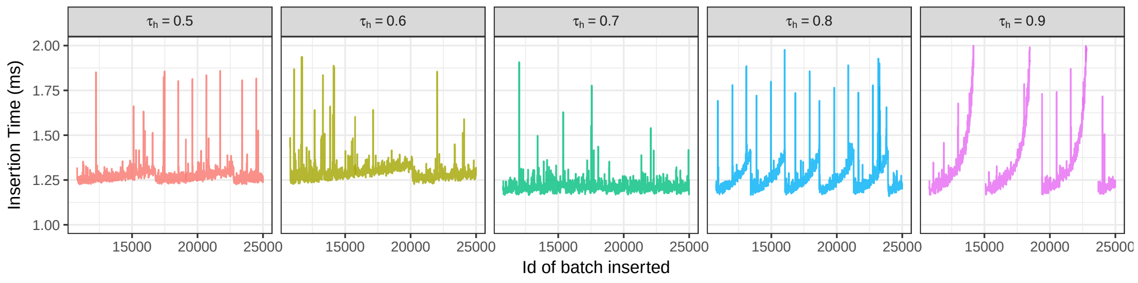


(a) Total average running time of the experiment.



(b) Average time of each bulk removal operation.

Figure B.10: PMQ performance at steady regime for different τ_h thresholds.



PMQ are 3X times faster than B-Tree and 5X faster than R-Tree.

B.7.3 Evaluating Bulk Deletions

In the case of streaming data, the memory available limits the storage of information. We used a stashing procedure to evict old data while receiving new incoming records. Data structures are in a *steady regime* if they cannot grow after inserting a given number of records. At this point, a bulk removal is triggered to remove a number of the oldest elements in the data structure, given by the threshold parameter λ. How often removals are triggered depends on the rate of the incoming stream and the threshold λ. We keep the incoming rate constant by inserting a batch of 1000 elements at each simulation step. Because the bulk removal is slower than regular insertions, the choice of λ has an impact in two performance indicators: the average execution time of each operation (B.9(a)) and the bulk removal execution time (B.9(b)). To evaluate the indicators and choose the best λ, we insert in each structure a dataset of nearly 46 million elements and set the maximum number of elements to be stored to half of this size.

Figure B.9 shows running times for λ varying from 0.1% to 50% of the maximum capacity. For both the B-Tree and the R-Tree, the removal size has to be chosen carefully to balance the time spend on each removal operation and the total running time. If the removal size is large (over 3.13%), each operation deletes many elements at once causing expensive removal operations that take over 1 second (black horizontal line in B.9(b)). If the removal size is small (under 0.78%), each removal is fast since only a small percentage of elements are evicted. However, it increases the frequency of removals impairing the total running time (B.9(a)). In opposite, the PMQ triggers element removal automatically when the top density threshold is reached. As a consequence, the execution time of removals is much less sensitive to λ. As B.9(b) shows, for any removal size, the running time is under one second. As expected, the total running time for all structures is best with larger and less frequent removals (B.9(a)). Therefore the choice of the parameter λ should favor larger removal sizes.

In Table B.1, we summarize the results from B.9(a) and B.9(b). We show, for each structure, the best tradeoff between removal and average running time. The B-Tree and R-Tree require a small removal percentage (λ), while PMQ removes 50% of if elements and it is $2\times$ faster for both removal and average running time. In Table B.2, we take the λ value that gives the best performance tradeoff for the B-Tree and set it to the other data structures. Once again, PMQ performs best and is the only one to make removal operations in less than a second.

Table B.1: Parameter λ set for the best relation of removal *RM* time and average *Avg* runnning time for each algorithm.

Algo.	Min. Elts	RM (ms)	Avg. Run Time (ms)	RM Interval	λ
B-Tree	22.7 M	1331	2.19	735	3.13%
PMQ	11.7 M	**550**	**1.09**	11744	50%
R-Tree	23.1 M	1287	4.43	368	1.57%

In Figure B.10 we compare several values of τ_h threshold (with fixed $\tau_0 = 0.92$) for the PMQ at steady regime: when the PMQ density reaches τ_h a bulk removal is triggered keeping at least 10.8 M elements. The value $\tau_h = 0.7$ gives the best average insertion time. The PMQs with $\tau_h = 0.5$ and 0.6 require twice more storage memory compared to the ones with 0.7, 0.8 and 0.9, with a high average insertion time. High τ_h values (0.8 and 0.9) leave the PMQ fill, leading to costly rebalances of large windows. The value $\tau_h = 0.7$, chosen for all our other experiments, gives the best average insertion time with low memory footprint. A high value $\tau_0 = 0.92$ gives the best results, allowing some local high-density spots.

B.7.4 Evaluating the Rebalancing Procedure

PMQ supports a rebalancing procedure that is only activated when necessary. In Table B.3 we simulate a tweet insertion rate of 1000 tweets per second. We present the average insertion time in PMQ after reaching a steady state (*i.e.* after the first top-level rebalance that started removing tweets). Notice that during this steady state, elements deletion neither leads to halving nor doubling the PMQ size. Between the top two level rebalances, the number of elements in the container varies from $Elts\_min$ to $Elts\_max$. The maximum value in the table corresponds to a single insertion that triggers a top-level rebalance. Although these periodic rebalances can take up to one second, this latency is hidden from the user as the mean insertion time (and the 99th percentile) is much smaller than the insertion rate.

We also evaluated how PMQ scales with varying insertion rates. We used a time window of 6 hours and increase the insertion rate up to 8k tweets/s. Figure B.11 shows the average insertion time and standard deviation. PMQ takes less than $8\ ms$ to digest 6000 tweets per second, the current average number of tweets posted per second worldwide (STATS, Mars 2017).

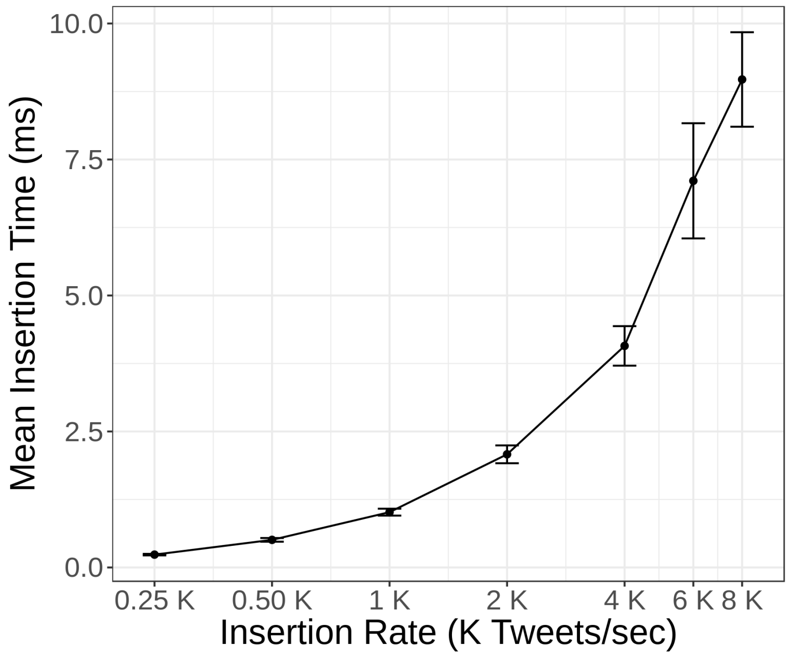
Table B.2: Comparison using same λ optimized for the B-Tree.

Algo.	Min. Elts	RM (ms)	Avg. Run Time (ms)	RM Interval	λ
B-Tree	22.7 M	1331	2.19	735	3.13%
PMQ	22.7 M	**601**	**1.80**	735	3.13%
R-Tree	22.7 M	1984	3.60	735	3.13%

Table B.3: Insertion time of batches of 1K elements in a PMQ with different time-windows λ. The number of elements in the container varies from Elts_min to Elts_max. The Mean, 99% and Max times are in ms.

λ	Elts_min	Elts_max	Mean	99%	Max
$3h$	$10.8 * 10^6$	$11.74 * 10^6$	1.209	1.066	265.613
$6h$	$21.6 * 10^6$	$23.48 * 10^6$	1.310	1.134	554.971
$9h$	$32.4 * 10^6$	$46.97 * 10^6$	1.278	1.587	1007.040
$12h$	$43.2 * 10^6$	$46.97 * 10^6$	1.423	1.321	1045.950

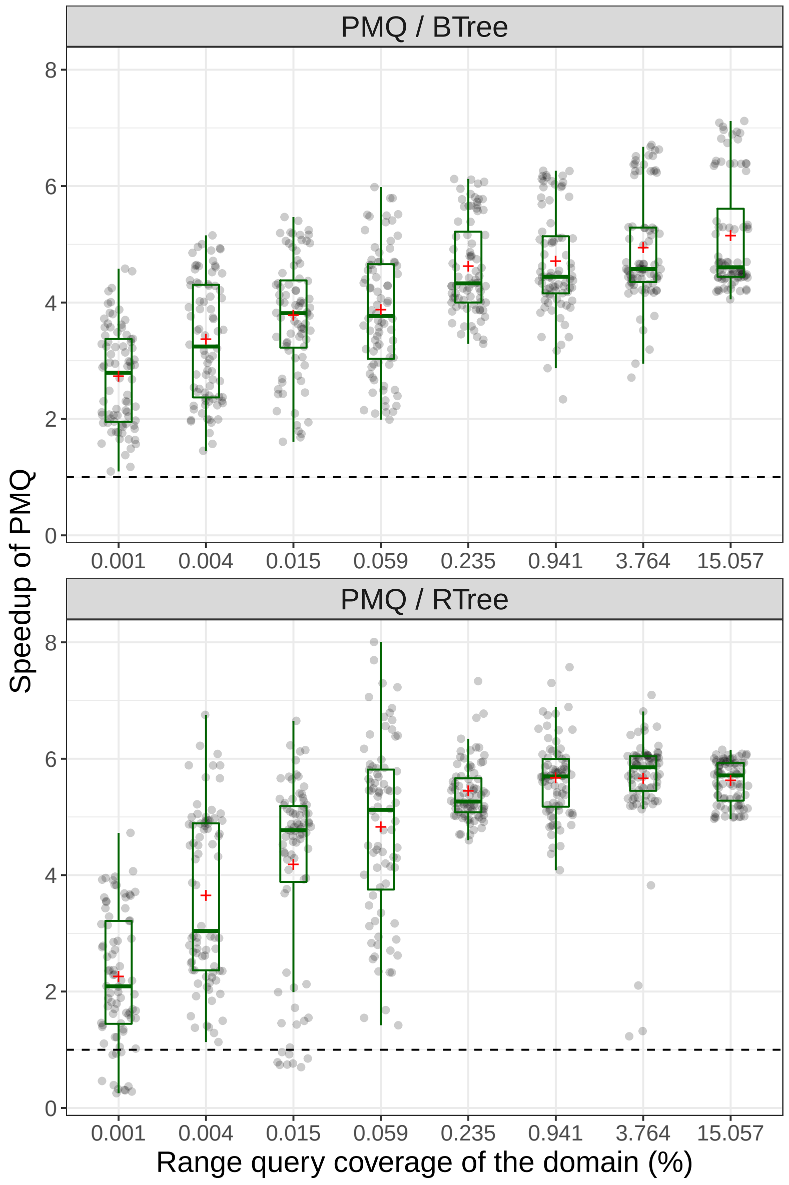
Figure B.11: PMQ average insertion time with a window of $6h$ and varying rates. Current Twitter insertion rate (6K tweets/s) can be processed under 7.5 ms.



B.7.5 Evaluating Range Queries

We evaluated the range query performance of the different data structures. We defined synthetic queries at varying sizes and different positions to simulate searches over the world map. Queries are defined using latitude and longitude coordinates over a rectangular map of the world using the Mercator projection. Each query is specified by its center latitude and longitude coordinates (lat, lon), where $lat \in \Re_{-90,+90}$ and $lon \in \Re_{-180,+180}$, and by its width $W = \{w \in \Re : w = \frac{90}{2^i} \wedge 0 \leq i < 8\}$. For each w in W, we randomly pick ten tweet coordinates from the dataset to generate a unique query. We discarded queries not fully contained on the world map. As a result, the query dataset

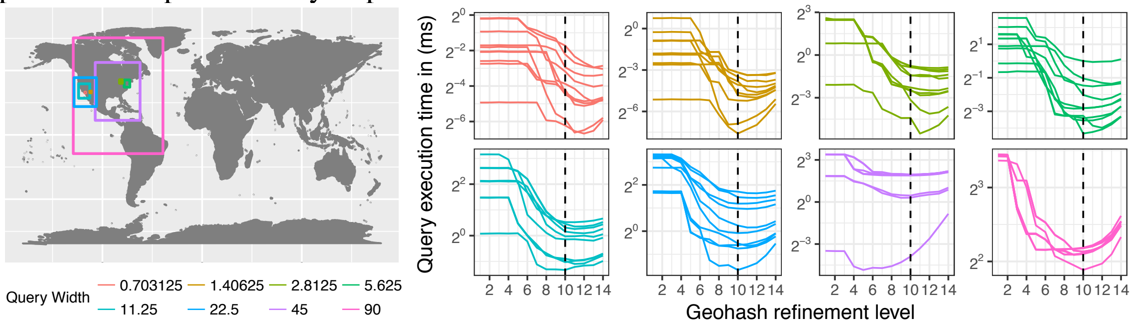
Figure B.12: Range queries: PMQ speedup over the B-Tree and R-Tree. Each boxplot represents the speedup of throughput for each query instance. The average speedup is denoted by red crosses. PMQ is faster than the B-Tree in all cases. Compared to the R-Tree, PMQ has a speedup on 97% of queries tested. The cases where PMQ performs worst corresponds to queries returning a small number of elements compared to the dataset size.



we built has 80 queries. This set of queries was run over 8 different datasets at varying sizes from 1M to 128M elements, for a total of 640 query results. The size of the dataset is given by a parameter S defined as $\{1M \times 2^i : 0 \leq i < 8\}$. We performed each query 10 times and computed the average running time. Since we fixed the number of elements in memory, we computed the throughput of each query as the number of records returned by the query divided by the running time (in ms).

We compared the throughput by showing the speedup of PMQ over B-Tree and R-Tree (Figure B.12). The boxplots in Figure B.12 show the speedup grouped by w.

Figure B.13: Examples of Range Queries. We define 8 different query widths. The B-Tree and PMQ use 10 levels of quadtree refinement for range queries. We choose this parameter experimentally to provide the best overall results.



The labels on the x-axis show the range query coverage relative to the total area of the domain. PMQ and B-Tree use the same querying mechanism based on the recursive space partitioning of a quadtree. In this case, PMQ is always better than the B-Tree, with speedups that can achieve up to 7× on the largest range queries. The speedup over the B-Tree is proportional to the number of elements returned by the query and is mainly due to the memory locality of PMQ. When the number of elements scanned to answer a query increases, the B-Tree has to access nodes scattered in memory locations, thus causing a poor usage of the cache memories. The average speedup increases with the range query size.

Since the R-Tree is a pointer-based data structure, its internal nodes have bounding boxes containing the space occupied by its children, and only leaf nodes have the actual elements. Because of its internal index, the R-Tree is efficient for point queries, with good performance when a small amount of elements is queried. In a streaming dataset, elements are inserted individually in the data structure, as they arrive from the stream and without any specific ordering in memory. When the size of the range queries increases, the cost of scanning more elements hinders the throughput. The query algorithm uses a max depth parameter to limit the refinements done in the linear quadtree. The max depth of 10 used in this experiments was found to give the best results (for B-Tree and PMQ) as shown in Figure B.13. The refined quadrants that do not fall entirely inside the queried region are scanned linearly to test the elements contained in the queried region. As a consequence, small range queries in regions with a high density of elements suffers from discontinuities in the Z-curve ordering. The throughput has a negative impact when the number of valid elements returned is low compared to the number of records scanned. Despite this, in our experiments the PMQ query algorithm outperforms the R-Tree (which

Figure B.14: Top-k Queries: cumulative percentages of query latency for $K = 100$, $R = 30$ km and $T = 10000$ seconds. We compared the search performance of PMQ against the *Kite* framework.

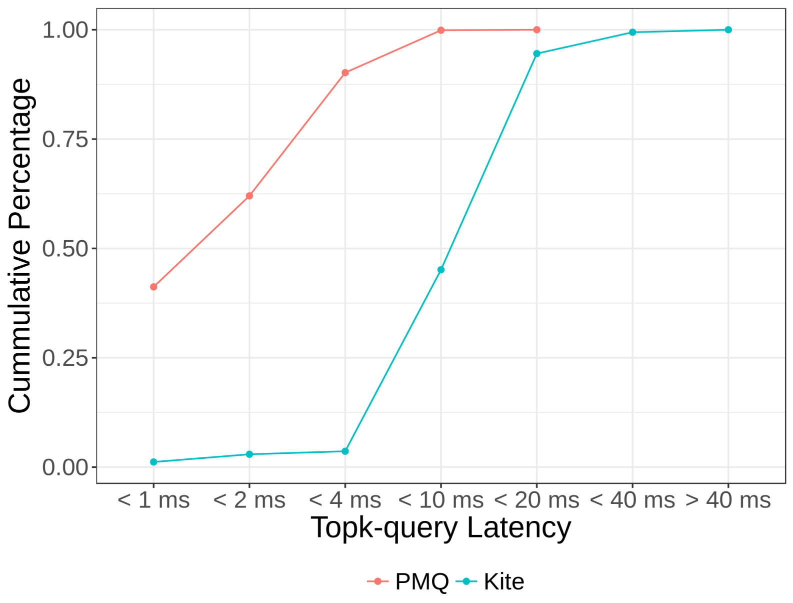
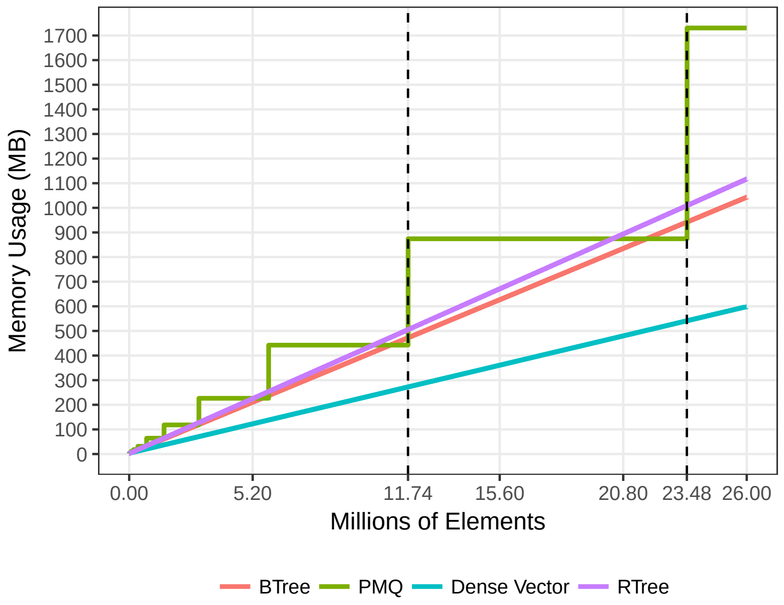


Figure B.15: PMQ memory usage depends on density thresholds $2\rho_h < \tau_h$. `Boost C++` R-Tree has a bigger index overhead than `stx::btree`.



does not rely on a Z-curve) by 5.5 times in 97% of the largest range queries.

B.7.6 Evaluating Top-k Queries

We compared the performance of top-k queries implemented on top of our PMQ against the `Kite` framework (MAGDY; MOKBEL, Mars 2017). We generated 10K top-

K queries from the check-in locations of the Brightkite social network (LESKOVEC; KREVL, Mars 2017). Queries correspond to users, in a given location, trying to find the most relevant tweets nearby.

At the moment of execution of the top-k queries, there were 10M tweets stored in PMQ. For each query, we measured the latency of accessing the storage array and computing the top-k elements. The top-k ranking function used the default parameters values $K = 100$, $R = 30km$ and $\lambda = 10000$ seconds. Temporal and spatial scores in PMQ were balanced with values 0.2 and 0.8 respectively. `Kite` does not offer a balancing parameter for the temporal and spatial dimensions, ranking elements with radius r merely according to the temporal dimension. Figure B.14 shows queries for different latency values. PMQ answers 90% of the queries in less than 4 ms while *Kite* can only process 3% of it. Kite uses a regular grid as a spatial index. Since the grid does not change to adapt to the complexity of the data, Kite does not perform well under scenarios of streaming datasets.

The data being sorted first based on their Z-index and next their timestamp, a pure time based query with no spatial constraint would need to scan all the elements of PMQ. We have seen that the PMQ shows a good scan performance (Fig. B.8(b)). But if a majority of requests of this type are expected it may be advantageous to index data based on their timestamp first.

B.7.7 Evaluating Memory Usage

We measured the amount of resident set size used by each data structure individually (e.g., the physical memory used by the process code and data). The `Linux Kernel` maintains a pseudo-file system directory for each running process. By parsing `/proc/self/statm` we have access to the current resident set size in pages. We multiply this value by the page size from `sysconf()` to obtain the amount of used memory in bytes. We set the record size to `16 bytes`, the minimum space required to store spatial and timestamp metadata. Each batch inserts `1000 records` per iteration. For every iteration, we measured the current resident set size used by the data structures. Memory usage in a dense vector is directly proportional to the number of records, as shown in Figure B.15, and serves as a baseline since this alternative does not have any storage overhead. PMQ memory usage depends on the max density parameter, τ_h (see subsection B.4.1). In our experiments τ_h is configured to 0.7, *i.e.*, the used memory slots

Table B.4: Memory usage summary

	Number of Elements stored	
Algorithm	11.7 M Elts	23.4 M Elts
Dense Vector	275.32 MB	545.97 MB
PMQ	882.97 MB	882.97 MB
B-Tree	477.00 MB	951.63 MB
R-Tree	510.42 MB	1019.15 MB

correspond at maximum to 70% of slots allocated. As the number of elements in the data structures increases, PMQ doubles its size when the maximum density is reached (note the staircase-shaped curve of PMQ in Figure B.15). In the experiments of subsection B.7.3, the maximum number of elements presented in memory was $23.488.000$, which corresponds to the memory consumption show in Table B.4.

B.7.8 Discussion of the Evaluation Results

The design of a data structure that is at the same time efficient for insertion/removal operations and large range queries requires careful analysis of trade-offs. Experiments have shown that PMQ offers a good tradeoff between both types of workloads. At a steady regime PMQ can perform efficient bulk removals, which are usually expensive in tree-based data structure because they require a full scan of the data.

The execution of queries in PMQ is substantially different from R-Trees because there are no index pointers to locate records. Instead the PMQ keeps data sorted based on Z-indices, and it suffices to use a fast range searching algorithm. We used the same Z-order in PMQ and B-Tree. However, as data is inserted and removed dynamically, a tree structure becomes fragmented in memory. PMQ avoids this issue by keeping the locality of its records along the Z-curve. Our experiments always verified that PMQ outperforms the B-Tree.

Some insertions in the PMQ can lead to higher execution times when a full rebalance is required: when the PMQ size needs to be doubled (Figure B.7), or, with a lesser impact, at steady regime during bulk data removals (Figure B.10). To mitigate the impact on query response time, one could rely on multithreading to overlap as much as possible rebalances with queries, adapting the approach proposed in (BENDER; DEMAINE; FARACH-COLTON, 2005) for the PMA.

B.8 Conclusion and Future Work

We introduced PMQ, a new data structure to keep sorted a stream of data that can fit in a controlled memory budget. PMQ reorganizes itself when needed with a low amortized number of data movements per insertion ($O(\log^2(N))$). Amongst the data structure compared, PMQ, B-Tree, and R-Tree, PMQ proved to have the best performance trade-off between insertion and searching times. Experiments showed that PMQ enables querying a continuously updated window with the latest arrived tweets in real-time. PMQ can maintain a significant amount of data in memory, filling the gap between stream processing engines working only on small windows of received stream, and other classical persistent storage solutions.

One direction for improvement would be to combine in-memory and persistent storage in a multi-level PMQ. The lazy stashing protocol might not adapt to some needs, as old data may stay a *long time* (up to the next top rebalancing) before being removed. We plan to develop a more reactive protocol for such situations. The current implementation imposes that every operation must acquire a thread lock before accessing or modifying PMQ. All requests are thus performed sequentially, which limits the number of transactions that PMQ can support.

B.9 Proof of the PMQ Amortized Cost

Let first identify an important property on windows densities after rebalance. A j-level window w_j is rebalanced when overfull ($d(w_j) > \tau_j$)). The rebalance occurs at the smallest underfull upper window w_l with $l > j$, i.e. the smaller one checking $d(w_l) < \tau_l$)). In worst case this is a top level rebalance requiring to double the PMQ size. After rebalance the density of w_j checks:

$$d(w_j) < \tau_l < \tau_j, \tag{B.6}$$

by Equation B.1 of page 148. So w_j gets a density $d(w_j) < \tau_{j+1}$.

Now let consider a window w_j and let see how many insertions are necessary in this window so that it triggers a rebalance, i.e. it requires to rebalance the parent window w_{j+1}. We assume w_j just get rebalanced, thus $d(w_j) < \tau_{j+1}$ by Equation B.6. The next rebalance triggered by w_j occurs once $d(w_j) > \tau_j$, i.e. after the insertion of

$$(\tau_j - \tau_{j+1})2^j K$$

elements where $K = O(\log(N))$ is the segment size.

Such rebalance requires to move $2^{j+1}K$ elements. If the rebalance occurs at the root window ($j = h$), the PMQ first makes a full scan of the PMQ to identify the data to be stashed. These data are next removed during the rebalance that is either performed on $w_h = O(N)$ if the new density is bellow τ_h or on a twice larger window after doubling the PMQ size. Thus a root rebalance cost is bounded by $2^{h+2}K$. We also need to count the cost of updating the accounting array. Each rebalance triggered by w_l leads to update $2^{j+2} - 1 + h - (j+1)$ elements of the accounting array.

Putting all these costs together, we have a cost associated to a rebalance triggered by w_j bounded by:

$$2^{j+2} + h - j - 2 + 2^{j+2}K < 2^{j+2}K + 2^{j+2} + \log(N).$$

This leads to the amortized cost per insertion of:

$$\begin{aligned} \frac{2^{j+2}K + 2^{j+2} + \log(N)}{(\tau_j - \tau_{j+1})2^j K} &< \frac{4K + 4 + \log(N)}{(\tau_j - \tau_{j+1})K}, \\ &= O(\log(N)) \end{aligned}$$

by Equation B.2 of page 148.

When an element is inserted into the PMQ, it actually contributes to the density of all enclosing windows from the segment up to the root, i.e. of $h = O(\log(N))$ windows. The amortized rebalance cost per insertion into the PMQ is thus $O(\log^2(N))$.

Each element needs to be inserted in the right place in the PMQ. If inserted one by one a binary sort is used with cost $O(\log(N))$. If inserted by batches, the insertion array is sorted with a cost per element that is also bounded by $O(\log(N))$. Added to the amortized rebalance cost, we get an unchanged total amortized cost per insertion of $O(\log^2(N))$.

Appendix C IMPLEMENTATION OF FOURIER SPECTRUM-BASED APPROACH TO REPRESENT DECISION TREES

```
import math
import matplotlib.pyplot as plt
import numpy as np


def add1(t, modules):
  stop = False
  cnt = len(t) - 1
  carry = 0
  t[cnt] = t[cnt] + 1
  while (not stop) and (cnt >= 0):
    t[cnt] = t[cnt] + carry
    if t[cnt] == modules[cnt]:
      t[cnt] = 0
      carry = 1
      cnt = cnt - 1
    else:
      stop = True
  return t


def product(l):
  p = 1
  for i in l:
    p *= i
  return p


def invertDict(x):
  y = {}
  for key in x:
    y[x[key]] = key
  return y


def factory(x):
  def temp(y):
    return x * y
  return temp


def fEquals(f, g, modules):
  numDimensions = len(modules)
  currentTuple = [0 for i in xrange(numDimensions)]
  numCoefficients = product(modules)
  for i in xrange(numCoefficients):
    if not (f(currentTuple) == g(currentTuple)):
      return (False, currentTuple, f(currentTuple), g(currentTuple))
    currentTuple = add1(currentTuple, modules)
  return True


def binarizeChar(ch, sampleSpace, dimensionIndex):
  print sampleSpace.dictDataToNumber, dimensionIndex
  return sampleSpace.dictDataToNumber[dimensionIndex][ch]


def binarizeSchema(schema, sampleSpace):
  result = []
  for i, t in enumerate(schema):
    if t == '*':
      result.append('*')
    else:
      result.append(binarizeChar(t, sampleSpace, i))
  return result


class FourierBasisFunc:
  def __init__(self, signature, dimSizes, numDimensions):
    self.signature = signature
    self.dimSizes = dimSizes
    self.numDimensions = numDimensions

  def applyPartialFunc(self, schema):
    numDimensions = self.numDimensions
    pr = []
    print schema, self.signature
    for i in xrange(numDimensions):
      if schema[i] == '*' and self.signature[i] > 0:
        return 0
      elif schema[i] == '*':
        pr.append(0)
      else:
        pr.append(schema[i])
    print 'final pr', self.signature, pr
    return self.applyFunc(pr)

  def applyFunc(self, pr):
    totalSum = 0
    numDimensions = self.numDimensions
    for l in xrange(numDimensions):
      totalSum = totalSum + \
                 ((self.signature[l] * pr[l]) / (self.dimSizes[l] * 1.0))
    return complex(
      math.cos(
        2.0 *
        math.pi *
        totalSum),
      math.sin(
        2.0 *
        math.pi *
        totalSum))


class SampleSpace:
  def __init__(self, dictDataToNumber, dictNumberToData):
    self.dimSizes = [len(t.keys()) for t in dictDataToNumber]
    self.dictDataToNumber = dictDataToNumber
    self.dictNumberToData = dictNumberToData

  def numDimensions(self):
    return len(self.dimSizes)

  def possibleValues(self, dimIndex):
    return self.dictDataToNumber[dimIndex].keys()

  def size(self):
    totalSize = 1
    for i in self.dimSizes:
      totalSize = totalSize * i
    return totalSize

  def fourierBasis(self, signature):
    numDimensions = self.numDimensions()
    return FourierBasisFunc(signature, self.dimSizes, numDimensions)

  def toFourier(self, f):
    numCoefficients = self.size()
    numDimensions = self.numDimensions()

    currentTuple = [0 for i in xrange(numDimensions)]
    coefs = {}

    for i in xrange(numCoefficients):
      coeff = complex(0, 0)
      basisFunction = self.fourierBasis(currentTuple)
      xx = [0 for _id in xrange(numDimensions)]

      for j in xrange(numCoefficients):
        # print x,basisFunction(x),f(x)
        coeff = coeff + basisFunction.applyFunc(tuple(xx)) * f(xx)
        xx = add1(xx, self.dimSizes)

      coefs[tuple(currentTuple)] = coeff / (1.0 * numCoefficients)
      currentTuple = add1(currentTuple, self.dimSizes)
    return coefs


class DTree:
  def __init__(self, sampleSpace, schema):
    self.sampleSpace = sampleSpace
    if len(schema) == 0:
      self.schema = ['*' for t in xrange(sampleSpace.numDimensions())]
    else:
      self.schema = schema
    self.decisionVarIndex = -1
    self.children = {}
    self.label = None

  def __call__(self, pt):
    return self.applyNormalizedFunc(pt)

  def applyFunc(self, pt):
    numChildren = len(self.children.keys())
    if numChildren == 0:
      return self.label
    else:
      valueToFollow = pt[self.decisionVariable]
      return self.children[valueToFollow].applyFunc(pt)
```

```
def applyNormalizedFunc(self, pt):
  numChildren = len(self.children.keys())
  if numChildren == 0:
    return self.label
  else:
    numDimensions = len(pt)
    originalPt = []
    for i in xrange(numDimensions):
      originalPt.append(self.sampleSpace.dictNumberToData[i][pt[i]])
    return self.applyFunc(originalPt)

def build(self, data, labels):
  pass

def setSchema(self, newSchema):
  self.schema = newSchema

def splitField(self, fieldIndex):
  self.decisionVariable = fieldIndex
  self.children = {}
  numValsInDimension = self.sampleSpace.dimSizes[fieldIndex]
  for i in xrange(numValsInDimension):
    schema = list(self.schema)
    schema[fieldIndex] = self.sampleSpace.dictNumberToData[fieldIndex][i]
    child = DTree(self.sampleSpace, schema)
    self.children[self.sampleSpace.dictNumberToData[fieldIndex][i]] = child

def order(self):
  numStars = 0
  for ch in self.schema:
    if ch == '*':
      numStars
  return numStars

def numInstances(self):
  numInstances = 1
  numDimensions = len(self.schema)
  for i in xrange(numDimensions):
    ch = self.schema[i]
    if ch == '*':
      numInstances = numInstances * self.sampleSpace.dimSizes[i]
  return numInstances

def numInstances(self):
  numDimensions = len(self.schema)
  totalSize = 1
  for i in xrange(numDimensions):
    if (self.schema[i] == '*'):
      totalSize = totalSize * self.sampleSpace.dimSizes[i]

  return totalSize

def toFourier(self):
  numCoefficients = self.sampleSpace.size()
  numDimensions = self.sampleSpace.numDimensions()
  #
  currentTuple = [0 for i in xrange(numDimensions)]
```

```
basisFuncs = {}
coefs = {}

for i in xrange(numCoefficients):
  basisFuncs[tuple(currentTuple)] = FourierBasisFunc(
    currentTuple, self.sampleSpace.dimSizes, numDimensions)
  coeff = complex(0, 0)
  nodesToProcess = [self]
  while len(nodesToProcess) > 0:
    node = nodesToProcess[0]
    del nodesToProcess[0]
    # if currentNode is a leaf
    if node.label is not None:
      translatedSchema = binarizeSchema(
        node.schema, self.sampleSpace)
      coeff = coeff + (1.0 / numCoefficients) * node.numInstances() \
              * node.label * basisFuncs[tuple(currentTuple)] \
                .applyPartialFunc(translatedSchema)
    else:
      nodesToProcess = nodesToProcess + \
                       [node.children[key] for key in node.children]

  coefs[tuple(currentTuple)] = coeff
  currentTuple = add1(currentTuple, self.sampleSpace.dimSizes)
return coefs
```