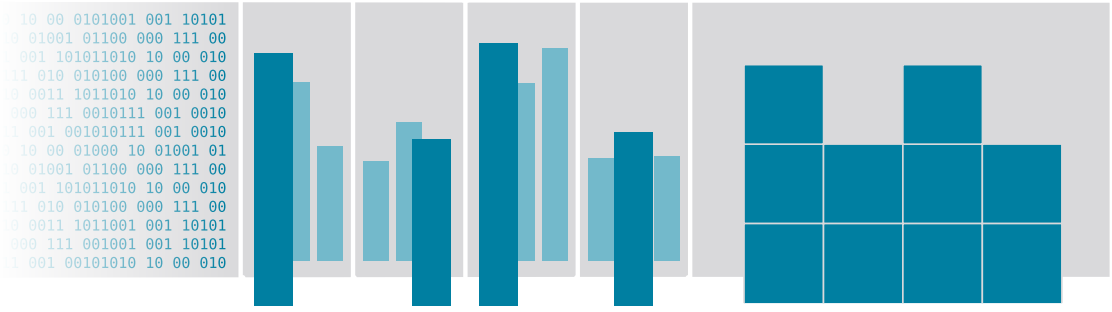
Computer Science Dissertation - Technische Universität Berlin

Uwe Jugel

Visualization-Driven Data Aggregation

Rethinking data acquisition for data visualizations



Visualization-Driven Data Aggregation

Rethinking data acquisition for data visualizations

vorgelegt von
Dipl.-Inf.
Uwe Jugel
geb. in Dresden

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender	Prof. Dr. Ziawasch Abedjan
Gutachter	Prof. Dr. rer. nat. Volker Markl
Gutachter	Prof. Dr.-Ing. Wolfgang Lehner
Gutachter	Prof. Dr.-Ing. Stefan Tai

Tag der wissenschaftlichen Aussprache: 13. April 2017

Berlin 2017

Dissertation

Visualization-Driven Data Aggregation

Rethinking data acquisition for data visualizations

Uwe Jugel

Abstract

Visual analysis of high-volume numerical data is traditionally required for understanding sensor data in manufacturing and engineering scenarios. Today, the visual analysis of any kind of big data has become ubiquitous and is a most-wanted feature for data analysis tools. It is vital for commerce, finance, sales, and an ever-growing number of industries, whose data are traditionally stored in relational database management systems (RDBMS).

Unfortunately, contemporary RDBMS-based data visualization and data analysis systems have difficulty to cope with the hard latency requirements and high ingestion rates required for interactive visualizations of big data. Disregarding the spatial properties of the visualization, they are incapable of effectively sampling or aggregating the big data for subsequent data visualization. The resulting big data visualizations suffer from measurable and perceivable visualization errors. Moreover, existing visualization-related techniques for data reduction are domain-specific and focus on a few custom types of visualizations. The underlying problems have neither been analyzed comprehensively and in general, nor in the context of RDBMS-based systems in particular. A general-purpose solution for visualization-related data reduction has been missing.

To facilitate truly interactive visualizations of the growing volume and variety of big data, computer systems need to change the way they acquire data for the purpose of data visualization. Therefore, the present work introduces the *Visualization-Driven Data Aggregation* (VDDA) that facilitates high-quality to error-free visualizations of high-volume datasets at high data reduction rates. Built on an in-depth analysis of the underlying problem of visual aggregation, VDDA defines specific data reduction operators for the most common chart types and for chart matrices. For RDBMS-based systems in particular, these operators can be used at the query level in a transparent query rewriting system, making VDDA applicable to any visualization system that consumes data stored in relational databases.

Using real-world datasets from high-tech manufacturing, stock markets, and sports analytics domains, this work demonstrates the applicability of VDDA, reducing data volumes and query answer times by up to two orders of magnitude, while preserving pixel-perfect visualizations of the raw data.

Zusammenfassung

Eine visuelle Analyse großer numerischer Datenmengen ist herkömmlich notwendig, um Sensordaten aus Konstruktions- und Fertigungsprozessen auszuwerten und zu verstehen. Heute ist die visuelle Analyse jeglicher Art von großen Datenmengen (Big Data) allgegenwärtig und eine meistgesuchte Funktion von Softwareprodukten zur Datenanalyse. Sie ist unverzichtbar für den Handelsverkehr, das Finanzwesen, den Einzelhandel und einer ständig wachsenden Anzahl an Branchen, deren Daten herkömmlich in relationalen Datenbankmanagementsystemen (RDBMS) gespeichert werden.

Leider haben gegenwärtige RDBMS-basierte Datenvisualisierungs- und Datenanalysesysteme Schwierigkeiten mit den hohen Anforderungen an Latenzzeiten und den hohen Dateneingangsraten fertig zu werden. Sie missachten die räumlichen Eigenschaften der Visualisierungen und sind dadurch nicht imstande, die für die Datenvisualisierung benötigten Stichproben und Aggregationswerte der großen Datenmengen effektiv zu erzeugen. Die resultierenden Datenvisualisierungen haben mess- und sichtbare Visualisierungsfehler. Bestehende Methoden zur visualisierungsbezogenen Datenreduktion sind anwendungsspezifisch und werden nur auf einige wenige, individuelle Visualisierungstypen angewandt. Die zugrunde liegenden Probleme wurden weder umfassend und allgemein, noch mit speziellem Bezug zu RDBMS-basierten Systemen analysiert. Es existiert keine universelle Methode zur visualisierungsbezogenen Datenreduktion.

Um zukünftig interaktive Visualisierungen der vielfältigen und wachsenden Datenmengen zu ermöglichen, müssen Computersysteme die Art der visualisierungsbezogenen Datenerfassung anpassen. Die vorliegende Arbeit entwirft dafür eine visualisierungsgesteuerte Datenaggregation (VDDA, engl. Visualization-Driven Data Aggregation), die einerseits hohe Datenreduktionsraten und andererseits Visualisierungen von sehr hoher Qualität ermöglicht. Aufbauend auf einer detaillierten Analyse des Problems der visuellen Aggregation, bietet VDDA spezifische Datenreduktionsoperatoren für alle gängigen Typen von Diagrammen und für Diagramm-Matrizen. In RDBMS-basierten Systemen können diese Operatoren dann für eine Datenreduktion auf Anfrage-Ebene angewandt werden, und sie ermöglichen dadurch eine transparente Anpassung der ursprünglichen visualisierungsbezogenen Anfrage. Die entwickelte Methode ist allgemein anwendbar und bietet eine verbesserte Datenerfassung für jegliche Art von Datenvisualisierungssystemen, die Daten aus relationalen Datenbanken beziehen.

Die Arbeit demonstriert die Anwendbarkeit von VDDA anhand realer Daten aus den Bereichen der Hochtechnologie-Fertigung, Aktienmärkte und Sport-Datenanalyse. Die übertragenen Datenvolumen und Antwortzeiten werden dabei um bis zu zwei Größenordnungen reduziert, während die Datenvisualisierungen pixelgenau erhalten bleiben.

Disclaimer

I declare that the content of the present dissertation is based on my own thoughts and research. To the best of my knowledge, I listed and referenced in this document all sources and tools used to develop the dissertation.

Acknowledgments

The idea for this work arose during my first year as Ph.D. candidate at SAP Research Dresden and the Database Systems and Information Management Group at TU Berlin. I would like to thank Prof. Dr. rer. nat. Volker Markl for supervising this dissertation, for the many suggestions and discussions, and for relentlessly pushing for quality rather than quantity. Furthermore, I would like to thank Prof. Dr.-Ing. Wolfgang Lehner and Prof. Dr.-Ing. Stefan Tai for their helpful advice and for acting as second and third reviewer for this dissertation.

During my studies, I got a lot of support from my colleagues at SAP. In particular, I would like to thank Dr. Zbigniew Jerzak and Dr. habil. Gregor Hackenbroich. They significantly contributed to the progress of my work through many discussions, professional, scientific, and methodological advice, and their co-authorship in my publications.

Special thanks also go to Dr. Anja Jugel for validating and improving the formal and orthographic quality of the thesis. She helped a lot in improving the figures and making complicated paragraphs more understandable.

I cordially thank my family, my friends, and especially and again my beloved wife Anja, who provided me the freedom and time required to conduct the endeavor of a dissertation. Her contribution to the progress of this work cannot be underestimated, since she took a lot of work off my hands, caring for our two children, who were both born during the course of the dissertation.

Uwe Jugel, September 2016

Contents

Publications

This dissertation extends, refines, and reuses previously published ideas, algorithms, passages, and figures from the following original works.

Peer-Reviewed Articles

[1] Uwe Jugel, Zbigniew Jerzak, and Volker Markl. Big data on a few pixels. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 895–900, 2016.

[2] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. VDDA: Automatic Visualization-Driven Data Aggregation in Relational Databases. *The VLDB Journal*, 25(1):53–77, 2015.

[3] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: A Visualization-Oriented Time Series Data Aggregation. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014. (best paper[1]).

[4] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. Faster Visual Analytics through Pixel-Perfect Aggregation. *Proceedings of the VLDB Endowment*, 7(13):1705–1708, 2014.

[5] Uwe Jugel and Volker Markl. Interactive Visualization of High-Velocity Event Streams. *Proceedings of the VLDB Endowment*, 5(13), 2012. (PhD Workshop paper).

[6] Uwe Jugel and Zbigniew Jerzak. Visualization-Driven Data Aggregation. In *CEUR Workshop Proceedings of LWDA 2016*, volume 1670, page 6. Hasso Plattner Institute, 2016. (invited talk).

[1] This research paper received the *VLDB Best Paper Award* at the 40th International Conference on Very Large Data Bases, in Hangzhou, China, on Sep 03, 2014.

Patents

[7] Uwe Jugel, Zbigniew Jerzak, and Eric Peukert. Pixel-Aware Query Rewriting, May 2015. US Patent Application 14/289,421.

[8] Uwe Jugel. Distributing pre-rendering processing tasks, April 2015. US Patent 9,009,213.

[9] Uwe Jugel. Method and System for Generating Data-Efficient 2D Plots, Sept 2015. US Patent Application 14/635,787.

Notation

This work uses the following abbreviations, mathematical symbols, relational algebra expressions, and SQL query statements.

List of Abbreviations

APCA	Adaptive Piecewise Constant Approximation
API	Application Programming Interface
AVS	Advanced Visualization System
CMA	Correlated Maximum Aggregation
CPU	Central Processing Unit
CSV	Comma-separated value
CTE	Common Table Expression
DBMS	Database Management System
DDL	Data Definition Language
DVMS	Data Visualization Management System
DVS	Data Visualization System
GB	Gigabyte
HD	High Definition
HDD	Hard Disk Drive
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol

ISIN	International Securities Identifying Number
JSON	JavaScript Object Notation
MB	Megabyte
MIC	Market Identifier Code
MSE	Mean Squared Error
MSP	Merging of Small Partials
ODBC	Open Database Connectivity
OLAP	Online Analytical Processing
PAA	Piecewise Aggregate Approximation
PIP	Perceptually Important Points
PSNR	Peak Signal-to-Noise Ratio
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDP	Ramer-Douglas-Peucker Algorithm
ReuWi	Reumann-Witkam Algorithm
SQL	Structured Query Language
SSD	Solid State Drive
SSIM	Structural Similarity
TB	Terabyte
TSDR	Time Series Dimensionality Reduction
UHD	Ultra HD (4k)
URL	Uniform Resource Locator
VDDA	Visualization-Driven Data Aggregation
VisWy	Visvalingam-Whyatt Algorithm

Mathematical Symbols

w	Chart width (horizontal pixel size of a visualization or matrix cell).
h	Chart height (vertical pixel size of a visualization or matrix cell).
W	Matrix width (horizontal pixel size of a chart matrix).
H	Matrix height (vertical pixel size of a chart matrix).
u	Number of matrix columns.
v	Number of matrix rows.
$\mathbb{R}_w$	Continuous horizontal value range $[0, w]$ of a visualization.
$\mathbb{R}_h$	Continuous vertical value range $[0, h]$ of a visualization.
$\mathbb{N}_w$	Discrete horizontal value range $\{1, 2, ..., w\}$ of a visualization.
$\mathbb{N}_h$	Discrete vertical value range $\{1, 2, ..., h\}$ of a visualization.
$\mathbb{N}^+$	Range of strictly positive integer numbers $\{1, 2, ..., \infty\}$.
$a \cdot b$	An interpunct $\cdot$ is used as arithmetic multiplication operator.
a/b	A forward slash / is used as arithmetic division operator.

Relational Algebra Notation

$T(t, v)$ Definition of a time series relation T, i.e., a set of unique records (t, v), with two attributes t and v.

$|T|$ Size of a relation T, i.e., number of records in T.

$\pi_t(T)$ Projection of an attribute t from a relation T.

$\pi_{t' \leftarrow f(t)}(T)$ Projection of a derived attribute t', computed using a transformation function f on each record in T.

$\sigma_\theta(T)$ Selection (filtering) of records, according to a boolean expression θ on one or several attributes of T, e.g., using $\theta \leftarrow t < 120$.

$Q = \sigma_\theta(T)$ Definition of a derived relation Q as specific query on a relation T.

$Q(T)$ Definition of a derived relation Q as unspecified query on T.

$|Q|$ Size of a relation Q, e.g., number of records in $Q = \sigma_\theta(T)$ or $Q(T)$.

$G_{f(v)}(Q)$ Aggregation of the values of attribute v over all records in Q, using the aggregation function f.

${}_{f_g(t)}G_{f_1(t),f_2(v)}(Q)$ Grouping aggregation of Q, using a grouping function f_g and computing aggregated values using f_1 and f_2.

${}_{k}G_{k \leftarrow f_g(t),f_1(t),f_2(v)}(Q)$ Grouping aggregation of Q that projects a computed group key k as an additional attribute.

$Q_1 \bowtie_\theta Q_2$ Join of two relations Q_1 and Q_2, e.g., $Q_1(t, v)$ and $Q_2(k, v_{max})$, based on a boolean expression θ on the attributes of Q_1 and Q_2, e.g., using $\theta \leftarrow f_g(t) = k \wedge v = v_{max}$.

Essential SQL Statements

Projection `SELECT t,v FROM T`

Selection `SELECT * FROM T WHERE t < 120`

Aggregation `SELECT max(v) FROM Q`

Grouping `SELECT k, max(v) FROM Q GROUP BY k`

Join `SELECT * FROM Q1 JOIN Q2 ON Q1.k = Q2.k AND v = v_max`

Union `SELECT * FROM Q1 UNION SELECT * FROM Q2`

Limiting `SELECT * FROM Q LIMIT 1000`

Common Table Expressions (subqueries defined after `WITH`)

```
WITH
Q1 AS (SELECT t,v FROM Q WHERE t < 120),   -- subquery Q1
Q2 AS (SELECT max(v) AS v_max FROM Q1)     -- subquery Q2
SELECT * FROM Q1 JOIN Q2 ON v = v_max      -- final query
```

1. Introduction

The central topic of this work is the reduction of large datasets for the purpose of data visualization. Large datasets, e.g., obtained from sensor networks, may comprise millions to billions of records that are subsequently stored in a central database. Data analysis and data visualization applications, accessing these data over bandwidth-limited networks, often have difficulty to cope with the large data volumes, suffering from long data transfer times, when analyzing or visualizing large fractions of the data.

Therefore, a common approach to facilitate analysis and visualization of large datasets is to conduct a reduction of the requested data and transfer only a reduced subset to the downstream application. In this regard, existing forms of data reduction already provide a good approximation of the original data for data analysis in general, but they often fail to provide reasonably reduced data subsets for the purpose of data visualization. The development of appropriate, visualization-related data reduction algorithms is a difficult problem, requiring to anticipate which records will eventually be visible on the screen and which can be omitted.

The primary goal of the present work is to develop a data reduction technique that particularly considers the spatial and perceptual properties of data visualizations on raster displays, such as the pixel width and height of the visualization. This chapter presents the background, motivation, and idea for the work, leading to the definition of the research goals. The chapter concludes with an overview of the solution and the structure of the thesis document.

1.1. Background and Motivation

Since the digital age started at the beginning of this century, the volume, velocity, and variety of "big data" [71] have been constantly increasing [56]. Today, thousands of shares are traded at stock markets in less than one second [15]. Hundreds of sensors in a single manufacturing machine report 100 to 1000 events per second per sensor [60, 62], acceleration sensors in sports wear acquire up to 2000 events per second [81], and smart meters of millions of customers

are shifting from monthly readings to 15 minute intervals [99, 61], resulting in billions of records to be managed in the attached databases. Similar data rates and resulting data volumes can be observed in an increasing number of scenarios, with data originating from social activity, sales numbers, trade processes, road traffic, network traffic, and many other sources. This work considers and henceforth denotes all of the above kinds of *high-volume* and *high-dimensional* data sources as "big data".

It is well understood that today's data volumes cannot be squeezed into our computer screens [95] without further treatment; even considering today's high-resolution displays with 3840×2160 or more pixels. Visualizing such large amounts of data effectively and efficiently is a challenging task. Specialized big data visualization systems solve this task by adhering to the much-cited *visual information seeking mantra*:

> "Overview first, zoom and filter, then details on demand." [94]

Thereby, existing interactive visualization systems allow the user to step-wise interact with the graphical representation of the data [53], relying on graphically configured dynamic queries to the database [93]. Dynamic queries at the visualization level facilitate a dynamic and iterative acquisition of data from the database. The corresponding database queries effectively filter the raw data, acquiring only those records that contribute to the desired visualization on the screen. Unfortunately, the filtered query results may still contain millions of records. For instance, considering only 10 hours of sensor data of a single common $100Hz$ sensor [60], already $10_{hours} \cdot 60_{minutes} \cdot 60s \cdot 100Hz = 3.6M$ records need to be transferred from the database. Some big data visualization systems already consider this cardinality of the query result and take additional measures to reduce the data, e.g., using online [16, 21] or offline aggregation [75] of the data.

However, with the prevalence of big data in many application domains, more and more common – non-specialized – visualization tools, such as spreadsheet programs, but also visualization frameworks [19], will inadvertently be the end point for large volumes of data. Ignoring the requirements for big data visualizations, an untrained user will often unwittingly try to visualize raw, unreduced datasets. The visualization tool may then gracefully refuse to visualize or acquire the data, leaving the user unsatisfied. In practice, even if the high-volume data can be acquired in a reasonable time, many visualization tools will – ungracefully – stop working, while trying to process millions of incoming records. In the worst case they consume all available system memory and freeze the user's operating system.

Besides upgrading the hardware and generally improving software efficiency, a data reduction is the only effective measure to significantly speed up data acquisition in data visualization systems. In this regard, commonly used data reduction techniques are the following.

- Limiting the number of records by truncating the query result to a specific number of records, starting at a specific offset of the result set.
- Compression of the query result [108, 74, 29, 58].
- Systematic or random sampling of the query result [26, 47, 18].
- Approximating the query result using average values [67, 68, 44, 13].

Unfortunately these generic and thus visualization-independent approaches have several drawbacks. They may either not be effective in decreasing the data volume, or they may result in defective visualizations, from which the viewers may draw wrong conclusions and make unfavorable business decisions.

In common data visualization systems, conducting a data reduction requires to run additional operations on the data, such as data aggregation in the database, data compression at the web server, and data decompression at the web client. Thereby, the processing overhead of a data reduction process must be compensated by the savings in network bandwidth and the savings in processing time at the downstream components.

To address these problems, previously proposed big data visualization techniques are primarily relying on specialized data management systems [16, 28] and often focus on one specific type of visualization [49, 21]. Previous works firstly neglect to define appropriate extensions of their data reduction algorithms to a broader range of visualizations, including many of the most common chart types [77], such as line charts and bar charts. Secondly, they do not generalize the considered data aggregation models to be applicable for commonly used database systems.

Even though it is well known, as titled by Ben Shneiderman [95], that visualizations of big data are “extreme visualization: squeezing a billion records into a million pixels”, current approaches to big data visualization fail to attain a high-level view of the general properties of big data visualizations. Entangled in custom solutions and wired to custom chart types, they overlook that there is an implicit, visualizations-inherent data reduction that has been conducted for decades but is not leveraged comprehensively, i.e., systematically for all kinds of visualizations.

The named data reduction is known as *overplotting*, occurring in data visualizations when individual data records are rendered as graphical marks to be eventually presented using one or more of the limited set of $w \times h$ available screen pixels. Thereby, the graphical marks and thus their pixels may often overlay, i.e., *overplot,* each other, so that two or more data records may be inadvertently represented by the same screen pixels. When rendering big data to a few pixels, many pixels will be overplotted hundreds to thousands of times.

Overplotting is seen rather as a problem than an opportunity for data visualizations. What is more, for the considered big data, e.g., with sensor signals recorded at up to $2000Hz$, overplotting becomes immanent for any visualization, even when displaying only a few seconds to minutes worth of data. The cardinality even of small fractions of the visualized big data is usually orders of magnitudes higher then the final number of data-representing pixels, e.g., the few thousand colored pixels on the primarily white canvas of a line chart.

In fact, the amount of overplotting, i.e., the effectiveness of data reduction at the pixel level, can be measured easily by counting all non-white pixels in a visualization and comparing them to the cardinality of input data. Table 1.1 lists the relevant numbers from three big data scenarios and depicts an overplotted line chart of each considered dataset. The depicted datasets are (a) the price of a share on the stock market, (b) the speed of a soccer ball [81], and (c) the electrical power of a sensor in a semiconductor manufacturing machine [60].

The obtained data reduction rates, i.e., comparing the number of drawn pixels to the number of data records, are $1:5$ to over $1:400$. Put otherwise, at least $80\% - 99.8\%$ of the data records are not visible in the final visualization of the data and should not have been acquired in the first place. Overplotting or a similar implicit or explicit visualization-related data reduction is inevitable for any kind of data visualization, since the cardinality of the input data often significantly exceeds even the total number $w \cdot h$ of available pixels. Consider for instance the $6M$ records of the soccer ball data or the $3.6M$ records of sensor data from the manufacturing machine, compared to the total number of $240 \cdot 240 = 57600$ pixels of the rendered images.

Overplotting is the underlying principle of the developed data reduction techniques, leading to the following core idea of the present work.

Core Idea. *Conduct the visualization-inherent, pixel-level data reduction as close to the data as possible.*

By selecting only the eventually visible records from the data, i.e., by constraining the requested data on the spatial properties of the considered visualization,

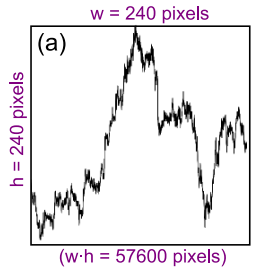
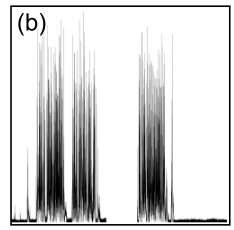
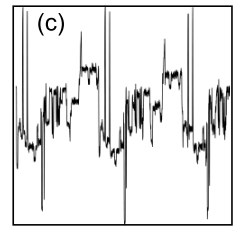
data source	share price (stock market)	ball speed (soccer game)	electrical power (manufact. machine)
cardinality	20000 (one day)	6M (2*30 minutes)	3.6M (10 hours)
line chart			
line pixels	4132 (7.2%)	16632 (28.9%)	8768 (15.2%)
white space	92.8%	71.1%	84.8%
data reduction	1:5	1:360	1:410

Table 1.1.: Visualization-inherent data reduction at the pixel level.

the acquired data will have a predictable maximum cardinality, independent of the number of underlying records. If conducted close to the data, e.g., as part of a visualization-related database query, such a visualization-driven data reduction has the potential to significantly speed up any subsequent data processing, including data transport, data transformation, and visualization rendering.

1.2. Goal of this work

The goal of this work is to redefine how computer systems acquire data for data visualizations. Data can no longer be acquired without verifying its cardinality. Most visualization-related queries to a database need to be altered to perform a data-centric reduction of the data. With massive datasets recorded pervasively in all kinds of scenarios, data reduction must be considered by any future visualization tool to remain competitive in the market.

Focusing on data stored and processed in relational databases, this work specifically considers the processing capabilities of such systems. Thereby, the developed solutions aim for generality by using the relational algebra [27] and the common data aggregation functions as the core operators, implemented in relational databases. The work moreover aims to be practical, by providing detailed implementation instructions for system builders, implementing

all considered data aggregation operators as queries in the industry standard *Structured Query Language* (SQL). Formally, the present work aims to solve the following research question.

Research Question. *How can the visualization-inherent data reduction be modeled as explicit data reduction, using the relational algebra with data aggregation?*

Solving the research question requires to identify and analyze the data reduction principles that are inherent in most common types of visualizations. This analysis will eventually facilitate the development of a data reduction framework that defines visualization-driven data reduction algorithms for the most common data visualizations, including basic charts, such as line charts and bar charts, but also chart matrices and space-filling visualizations [64].

Research Hypotheses. The two hypotheses of this work are the following.

1. *For all common types of visualizations, there is a data aggregation that models or approximates the data reduction inherent in the visualization.*

2. *The developed visualization-driven data reduction techniques are faster to compute and/or require less network bandwidth and/or provide a higher visualization quality than existing approaches.*

In summary, this work aims to provide a general yet easily adoptable visualization-aware data reduction model for all kinds of data visualization systems. Following the principles of this work, current and future visualization system providers will be able to enhance their products, featuring true visualization-awareness that leads to faster and more predictable response times for visualization-related queries and eventually to happier users.

Data Reduction Goals

The following formal and technical goals constitute a comprehensive set of requirements for visualization-aware data reduction methods in data visualization systems.

In data-intensive scenarios, network bandwidth is generally a much more valuable resource than memory bandwidth or CPU speed. High-volume, unreduced data must not leave the database, i.e., the number of records of a query result must be predictable. Therefore, the first goal for the developed data reduction techniques is the following.

Goal 1. *The data reduction must produce result sets of predictable size.*

To be able to efficiently compute the data reduction very close to the data, data reduction techniques must be implementable inside the database. Therefore, the data reduction may be implemented as user-defined function but should preferably be expressed using a higher-level declarative query language. This is summarized by the following second goal.

Goal 2. *The data reduction must be computable and optimizable by the database system.*

Reduced query results should comply to their original schema and data format, as expected by the visualization client. Otherwise, support for intermediate, e.g., compressed, data formats would have to be added to the visualization client. Such modifications and extensions are potentially expensive to develop and may require interfering with third-party software components. Thereof follows the third goal for data reduction.

Goal 3. *The data reduction must be transparent to the visualization client.*

Finally, the resulting visualizations should be correct by incurring only a low approximation error, i.e., the visualization of the reduced data should be the same or as similar as possible to the visualization of the originally requested raw data.

Goal 4. *Correct visualization-related data reduction techniques should result in the best possible approximating visualization of the raw data.*

The four formal goals are complemented with two additional performance goals, aiming for high data reduction rates and fast data processing.

Goal 5. *Fast data reduction techniques should involve only a low data processing overhead.*

Goal 6. *Effective data reduction techniques should provide high data reduction ratios.*

1.3. Solution Overview

The present work provides methods for data reduction in the context of data visualizations in general, with the data acquired from relational databases in particular. The following overview of the solution briefly describes the targeted system architectures and summarizes the underlying ideas and principles of this work.

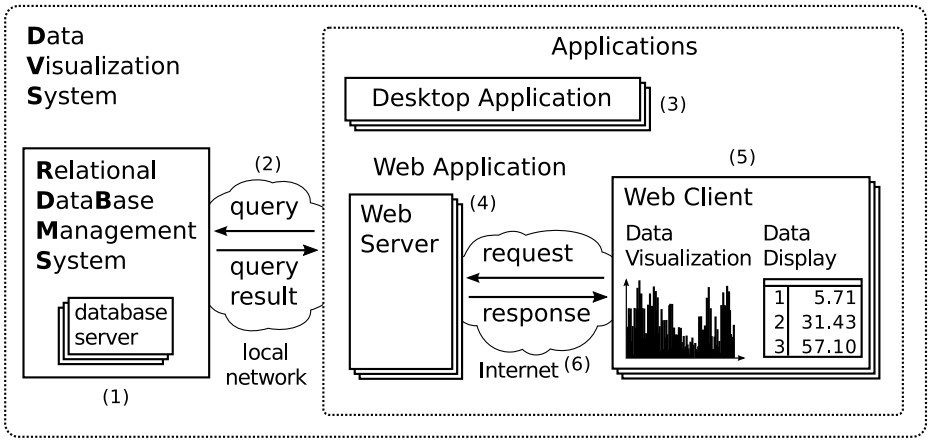


Figure 1.1.: Architecture of a data visualization system.

1.3.1. Considered Big Data Visualization System

To gain insight from big data, millions of stored records must be handled efficiently by a *database management system* (DBMS), aiming to provide fast data access and quick answers to analytical queries issued to the database. Thereby, raw data are commonly stored in a central database, e.g., hosted in a remote datacenter, and accessed over the network by connected software applications that present the acquired data and the results of analytical queries as interactive data visualizations to the user.

Figure 1.1 shows a high-level view of a computer system, composed of the different software components that this work considers to be used for storage, analysis, and visualization of big data. The considered system uses a *relational database management system* (RDBMS) (1) to securely store the recorded data. Applications can access the data over the network (2) and formulate analytical queries on the data. Applications can be monolithic desktop applications (3) or client-server systems, such as web applications, using a web server (4) and a web client, running in a web browser (5). The web client usually communicates with the web server over the Internet (6). This work denotes the entirety of the described and of similar end-to-end systems as *data visualization system* (DVS).

How quickly the requested data can be delivered from the database to the data visualization depends on the capabilities and resources of the data-processing components and on the network bandwidth between these connected

components. In particular, this work considers the following technical properties to influence answer times of visualization-related requests and queries.

- Query execution performance of the database system
- Database-outgoing network bandwidth in the local network
- Outgoing network bandwidth of the web server to the Internet
- Query post-processing performance of the application
- Rendering performance of the application

The volume of the requested data influences all five properties, with lower data volumes leading to faster query answer times; particularly if the data is reduced by one or more orders of magnitude. Therefore, if the user wants to quickly visualize a large dataset, i.e., without having to acquire the complete data, the DVS needs to select an appropriate subset of the data that can be used to visually represent the raw data. Which data subset to select depends on the type and structure of the desired visualization. This work aims to provide this data reduction by respecting the spatial and perceptual properties of data visualizations in the corresponding visualization-related queries.

1.3.2. Visual Aggregation for Basic Charts

This work formalizes the visualization-inherent data reduction at the pixel level as data aggregation at the query level. The developed data reduction technique is called *Visualization-Driven Data Aggregation* (VDDA).

The presented techniques and algorithms consider a visualization to be structured in single *display units*, such as the pixels of a scatter plot, the pixel columns of a line or bar chart, or the cells of a table. The number of vertical and horizontal display units is determined by the pixel size $w \times h$ of the visualization and the size of the display units, e.g, the width w_{bar} of a bar in a bar chart or the size of a table cell $w_{cell} \times h_{cell}$.

Any data visualization eventually displays a specific range of the data in each display unit, using a limited number of graphical marks. For instance, a single mark in a bar chart, i.e., a bar, can depict up to three attributes of at most one underlying data record, i.e., using the height, color, and label of the bar. The corresponding rendering process of a bar chart is illustrated in Figure 1.2. To render data to pixels, the raw data is transformed to a set of graphical marks, i.e., to geometric shapes that are subsequently rasterized to

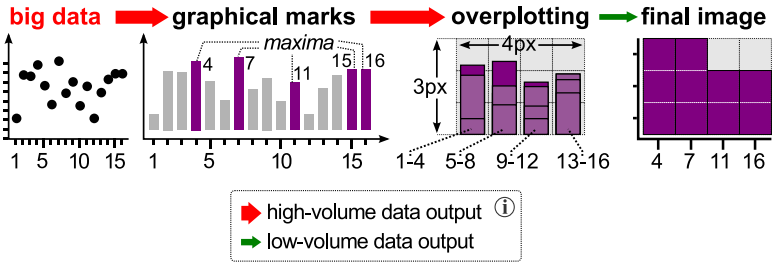


Figure 1.2.: Data reduction through projection of data records to pixels.

discrete pixels. Thereby, overplotting occurs, if several shapes or fractions of shapes are rendered to the same pixel, which is very likely for high-volume and dense data, e.g., with more than w records for a bar chart of $w \times h$ pixels. Assuming the records are rendered sequentially, the last shape for each display unit, e.g., for each pixel column in the bar chart, will often overplot most of the previously rendered shapes. For bar charts, the final image will effectively display all pixels of the last bar per pixel column and additionally several pixels of the longest bars that are longer than the last bar. If all bars have the same color, the final visualization of the raw data is effectively the same as the visualization of the *maxima* of the groups of records, corresponding to each pixel column. VDDA is based on such observations, defining for all common chart types a corresponding data aggregation operator that can be expressed in relational algebra and implemented as SQL query.

In the considered DVS (cf. Figure 1.1), VDDA is conducted as follows. First, the application encloses an original visualization-related query in a new data reduction query that simulates the pixel-level rendering process. The resulting rewritten query is then issued to the RDBMS instead of the original query. The actual data reduction is performed as close to the data as possible, by the query engine of the RDBMS. The new query result is a significantly reduced subset of the originally selected data. Thereby, VDDA works fully transparent to the visualization client because the original query is not modified but only enclosed in the data reduction query. The schema of the reduced query result does not differ from the schema of the original query result.

1.3.3. Advanced Visual Aggregation Techniques

VDDA, as described above, leverages the overplotting behavior in basic chart types only for a *single* data subset, such as the timestamps and values of one

single sensor. To acquire and reduce *multiple* data subsets, e.g., stored as database tables with multiple columns, each subset may be processed separately. The results are visualized using separate basic charts or subcharts in a chart matrix. The DVS then issues a separate query for each single subset.

However, multiple subsets can also be processed jointly to model the overplotting behavior of one subset over another subset. Therefore, this work does not only describe VDDA for the common basic chart types but also provides a detailed discussion of stacks and matrices of basic charts. In fact, VDDA covers all types of basic and composite visualizations, as found in common data analytics tools, i.e., as defined by Mackinlay et al. [77] for Tableau's automatic presentation system.

To define different VDDA operators for basic chart types and chart matrices, this work investigates and formalizes the *visual scalability* [32] of data visualizations. Thereby, the developed techniques not only respect the *spatial properties* but also additional *perceptual properties* of the visualization. In particular, this work shows how to incorporate established perceptual limits of data visualizations in the data reduction process, such as the maximum number of distinguishable colors [51, 107]. Combining the observed spatial and perceptual limits in a *spatio-perceptual capacity function* for each chart type, this work builds the basis for the following improvements of future visualization systems.

1. Automatic configuration of a chart matrix for displaying multiple data subsets using a perceptually optimal number of matrix cells.

2. Pruning of extent data subsets that exceed the visual capacity of a visualization.

Even though VDDA is primarily designed for continuous numeric data, this work also demonstrates how to incorporate categorical data and ordinal numeric data in the data reduction queries. As a result, VDDA can be used for data reduction in a broad range of application scenarios that exhibit interactive visualizations of big data.

1.4. Structure of the work

To answer the research question and confirm the research hypotheses, this work is structured as follows.

Chapter 2 first defines the basic terminology and related work in the research areas of database systems, computer graphics, and (big) data visualizations.

Thereafter, Chapter 3 describes how to generally conduct a visualization-driven data reduction in a data visualization system on top of a relational database management system by means of data-transparent query rewriting. The subsequent Chapter 4 provides a detailed discussion of different data aggregation models to be used by the query rewriting system, and moreover develops the M4 aggregation specifically for line charts. Chapter 5 then generalizes and extends the developed data aggregation model, defining a visualization-driven data aggregation that simulates or approximates the rendering process of each considered basic chart type and of chart matrices. Continuing the discussion on chart matrices and the visualization of multiple data subsets, Chapter 6 shows how to incorporate the perceptual limits of data visualizations for the purpose of automatic visualization and pruning. After evaluating the developed solution in Chapter 7, this work concludes with a summary of the contributions and an outlook on future work in Chapter 8.

2. Elements and Principles of Data Visualization Systems

In a data visualization system (DVS), many software components are used together to initially acquire, then transform, and eventually visualize the data. This chapter first describes the basic concepts, components, and tasks of a DVS. After explaining the general data processing model of a DVS, the chapter continues with a discussion of different DVS architectures, specifically of systems with relational databases that allow for secure storage and data-centric processing of the data to be visualized. The chapter concludes with a discussion of existing data reduction techniques and an overview of the types of data visualizations that are supported by the solutions developed for this work.

2.1. System Components

The following section complements the high-level description of the considered system architecture, introduced in Section 1.3.1 and Figure 1.1. Thereby, as depicted in Figure 2.1, this work relies on the following terms and principles and considers the following components and sub-components for data management and data visualization.

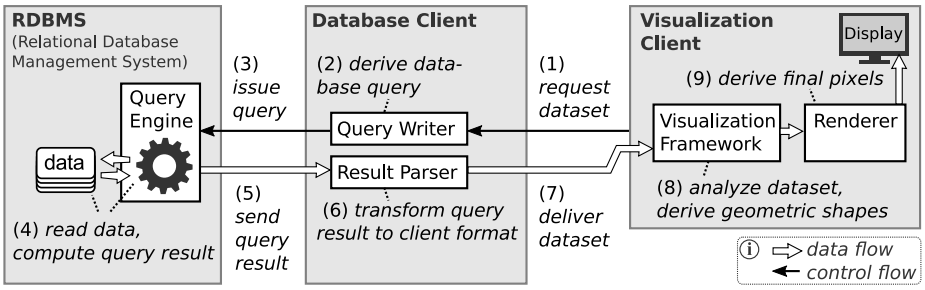


Figure 2.1.: Detailed DVS architecture and system behavior.

(Relational) Database. A database i.e., a database management system (DBMS), or a relational database, i.e., a relational database management system (RDBMS), persistently stores data on a physical medium. It provides access to the data through a query language and facilitates fast query processing, e.g., by parallelizing operations on the data and by temporarily storing datasets in the main memory of a computer system.

Database Client. A database client is a software component that automatically defines and issues – or provides a higher level interface to define and issue – queries to a database. For each issued query, the database client receives the query result from the database and provides it to downstream software components in a corresponding downstream data format.

Visualization Client. A visualization client is a software component that receives raw data or derived datasets from an upstream software component and displays the received data on an attached display, in the form of a specific data visualization.

These are the three main components of a data visualization system. In this work, the combination of a database client, the visualization client, and all subcomponents in-between these two are also denoted as *(database) application*. The entire system, including the DBMS and the application, is denoted as the *data visualization system* (DVS).

Upstream and Downstream

This work uses the terms *upstream* and *downstream* to indicate the direction of the data flow. An upstream component is a sender of the data, while a downstream component receives this data. Note that most software applications allow two-way communication, i.e., that upstream data may have been previously *requested by* a downstream component.

Data Sources

For the considered DVS, data sources are the raw data records stored in the database, e.g., the rows of a table in an RDBMS. How the stored data is acquired from *external*, upstream data sources like sensor networks is out of scope of this work. Most concepts and methods in this work are developed for raw data stored in an RDBMS, considering their downstream consumption in database applications. Throughout this work, data is supposed to be tabular

data with two or more *data columns*. Note that this work avoids the term (*data*) *dimension* to prevent confusion with the *layout dimensions* of a corresponding visualization.

Data and Control Flow

The data and control flows between the main components of a DVS on top of an RDBMS are illustrated in Figure 2.1 and comprise the following steps.

1. A visualization client requests a particular dataset from the database client.
2. The query writer of the database client converts this application-specific request to a database query.
3. The query is issued to the database.
4. The query engine of the database parses the query, derives an execution plan for the query, and computes the planned operations on the underlying data stored in the database.
5. The computed query result is sent downstream to the database client.
6. The result parser of the database client converts the data to the data format understandable by the visualization client.
7. The converted query result is delivered to the visualization client.
8. The visualization framework converts the data to a visualizable format.
9. The renderer traverses the derived visualizable data and computes the final pixels to be displayed.

The above steps define how the components of a visualization system communicate and illustrate the different stages of the requested data, until being visible on the screen. The named components are defined as follows.

Query Engine. The query engine is an essential part of a DBMS, optimizing and executing complex computations on the stored data.

Query Writer. The query writer reads several input parameters from a downstream component and uses these parameters to derive a query to be issued to the DBMS.

The query writer is particularly important for this work, since it is one of the components that can integrate the developed data reduction operators.

Result Parser. The result parser reads the query result from the database and converts it to a data format to be understood by the next downstream component.

For instance, many web applications will convert the query result to the JSON[1] format, which is supported natively in web browsers. Parsing and converting query results can be an expensive operation, especially if the parsed data volume is very large.

When the requested dataset is finally delivered to the visualization client, it needs to be further analyzed before being visualized. In particular, to define the numerical axes of a visualization, the minimum and maximum values of each data column of the dataset need to be computed. Similarly, additional metadata, such as the length of a dataset, may be required for the definition of the geometric primitives that are used to visualize the acquired records. The requested data or the derived geometry data, and the computed metadata are then sent to a downstream component for rendering them on a physical display or as a digital image. Note that some considered visualizations, e.g., line charts, do not require the computation of intermediate geometric primitives. The renderer may then traverse the original data directly and iteratively derive the resulting pixels of the visualization.

Visualization Framework. A visualization framework transforms datasets to data visualizations. The visualization framework analyzes the dataset to define the properties of the targeted visualization, such as the axes, legend, size, style, and padding, but also which drawing algorithms are used for rendering the data.

Renderer. The renderer provides a library of algorithms to convert numerical data in general and geometry data in particular to discrete pixels. A rendering algorithm defines for a given pixel matrix, how and which pixels are used to represent the underlying data points.

For instance, a line drawing algorithm [20, 24] defines for a given pixel matrix of $w \times h$ pixels and for each given line segment, representing each two consecutive records of the acquired data, which of the $w \times h$ pixels are *on the line* and need to be colored, and which pixels are *not on the line* and thus need to stay uncolored.

[1] JavaScript Object Notation, (json.org), ECMA Standard 404.

2.2. SQL Databases

This work primarily considers RDBMS for storage and provisioning of the data, and for running analytical queries on the data. Existing RDBMS implementations can easily handle gigabytes to terabytes of recorded data [35, 36]. Developed over decades, they provide unrivaled robustness and a variety of sophisticated tools for data analysis and data visualization. Built on top of declarative query languages, they are able to analyze, optimize, and parallelize a query, to provide fast query results for queries on large datasets. Modern relational databases have direct access to the data, i.e., direct memory access, and thus can read and conduct computations over millions of records within a few milliseconds [36].

Structured Query Language

The prevalent interface to access and analyze data in relational databases is the industry standard *Structured Query Language* (SQL) [10]. SQL is used by humans and computer systems alike. It is incorporated in most applications that allow to textually display or graphically visualize data in relational databases. For instance, many systems use dynamic queries [93, 95] that are configured interactively by the user of the DVS to eventually parametrize a corresponding SQL query. The query is then issued to the RDBMS to acquire the data to be visualized.

A declarative query language like SQL allows the user of a computer system to specify data operators on the stored records using a well-founded semantics. Therefore, SQL implements the relational algebra [27] and provides set operators and operators for filtering, projecting, transforming, and aggregating the data.

Query Optimization

SQL queries can be moreover optimized by all major RDBMS to speed up query processing [72, 96]. Therefore, the RDBMS may reorder the query operators or their predicates, e.g., the boolean logic expression of a `WHERE` or `HAVING` clause. By pushing down a filter predicate from an outer query to a nested subquery, large data volumes can often be effectively excluded from more expensive operators of the issued query. Additionally, the RDBMS may internally use different implementations of the query language operators [45], having dif-

ferent advantages and disadvantages, depending on the properties of the data, e.g., the data volume and the distribution of the incorporated data subsets.

By defining the proposed algorithms and data aggregation techniques primarily using the relational algebra and using optimizable, high-level SQL queries, this work aims for generality of the proposed solutions and allows the DVS to benefit from the available query optimization and query processing facilities.

2.3. Pixels on the Canvas

In most visualizations, there is a subtle difference between the user-defined size of a chart and the size of the canvas of the visualization.

Canvas. The canvas of a visualization is the drawing pane where the dataset is presented as a pixel matrix, using a limited number of discrete pixels. The canvas has a defined discrete width of w pixels and a height of h pixels. The canvas can at most display $w \cdot h$ pixels simultaneously.

Figure 2.2 conceptually depicts a complete visualization, i.e., a line chart of $w_{ext} = 14$ and $h_{ext} = 10$ pixels, including the chart's canvas and additional periphery that is commonly found in data visualizations. The 14×10 pixels of this line chart do not only comprise the pixels of the line segments to represent the data, but also the additional pixels required for the two axes and the padding of the chart. The canvas of the chart has an effective size of $w = 11$ and $h = 7$ pixels.

Axes, padding, borders and similar additional periphery are of less importance for this work, since the visual aggregation of the data occurs only on the canvas of a visualization. Unless stated otherwise, when discussing spatial and perceptual properties, and the components of a visualization, the remainder of this work conveniently uses "chart", "plot", or "visualization" conterminously to "canvas of" the chart, plot, or visualization.

Figure 2.2 moreover shows how the acquired data is projected to the continuous 2D space of the chart, i.e., the canvas of the chart. Thereby, the vertical and horizontal *data columns* must be projected to the corresponding vertical and horizontal *layout dimensions*. For instance, to project a set of records $\{(t_1, v_1), (t_2, v_2), ..., (t_n, v_n)\}$ with $t, v \in \mathbb{R}$, to the 2D space $[0, w] \times [0, h]$ of the visualization, the visualization client needs to determine the boundary values t_{min}, t_{max}, v_{min}, and v_{max} (cf. Figure 2.2a) and define a geometric transformation function for every layout dimension (cf. Figure 2.2c). Most common data

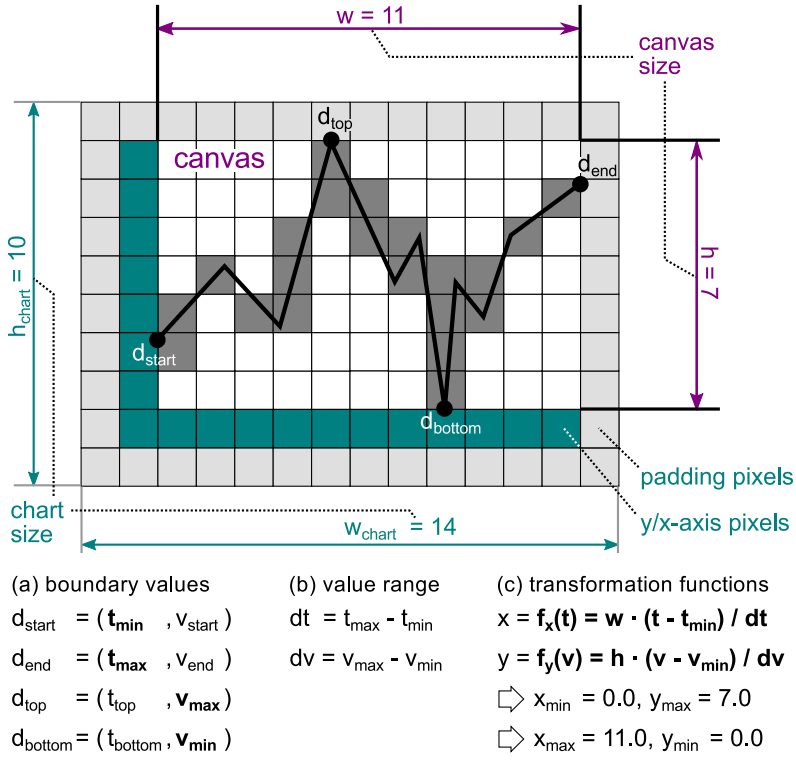


Figure 2.2.: Difference between chart size and canvas size.

visualizations conceptually use one or both of the following transformations functions f_x and f_y.

$$\begin{aligned} f_x(t) &= w \cdot (t - t_{min})/dt \\ f_y(v) &= h \cdot (v - v_{min})/dv \end{aligned} \tag{2.1}$$

These functions first shift the values t and v by the minimal values t_{min} and v_{min} into the value ranges $[0, dt]$ and $[0, dv]$, with $dt = (t_{max} - t_{min})$ and $dv = (v_{max} - v_{min})$. They subsequently divide the result by dt or dv, effectively projecting all values to $[0, 1]$. Eventually, they scaling up the representative values in $[0, 1]$ by w or h, to determine new representative values $x \in [0, w]$ or $y \in [0, u]$ for each original value t or v.

Thereby, the data or geometry data displayed on the canvas must not necessarily be restricted to using discrete values $x \in \{0, 1, ..., w\}$ or $y \in \{0, 1, ..., h\}$. A renderer may still incorporate the decimal fractions of the given numbers

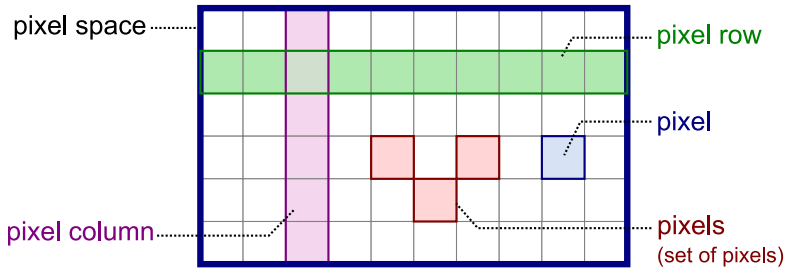


Figure 2.3.: Components of the pixel space.

of the *continuous* 2D space $[0, w] \times [0, h]$, i.e., for conducting an anti-aliased rendering of the data. However, at the last stage of the visualization pipeline, the data must eventually be converted to the discrete pixels of the *pixel space* of a visualization.

For representing the described continuous and discrete data ranges, the remainder of this work conveniently uses the notations $\mathbb{N}_w$ for the set of integer numbers $\{1, 2, ..., w\}$, $\mathbb{N}_h$ for the set of integer numbers $\{1, 2, ..., h\}$, $\mathbb{R}_w$ for the range of real numbers $[0, w]$, and $\mathbb{R}_h$ for the range of real numbers $[0, h]$. The continuous and discrete space of a visualization are then defined as follows.

2D Space. The continuous two-dimensional space $\mathbb{R}_w \times \mathbb{R}_h$ of the visualization canvas is denoted as the *2D space* of a visualization.

Pixel Space. The set of $w \cdot h$ points $\{(x_1, y_1), ..., (x_w, y_h)\}$, i.e., with $(x, y) \in \mathbb{N}_w \times \mathbb{N}_h$, represents the discrete pixels of the resulting image of a visualization and is denoted as the *pixel space* of a visualization.

The pixel space of a visualization is measured in pixels (px) and has a width of $w\,px$ and a height of $h\,px$ pixels. It can be decomposed into several components, illustrated in Figure 2.3 and defined as follows.

Pixel (entity). A pixel is the smallest spatial entity of the pixel space that can be displayed on a raster display. A pixel has a width and height of exactly $1px$ and an area of $1px^2$.

Pixel Column. A pixel column is a vertical slice of the canvas, having a width of $1px$, a height of $h\ px$, and containing all pixels of that slice from the top to the bottom pixel.

Pixel Row. A pixel row is a horizontal slice of the canvas, having a height of $1px$, a width of w px, and containing all pixels of that slice from the left-most to the right-most pixel.

The term "pixel" is commonly used for length measurement, area measurement, and area definition alike. To distinguish the different measures, this work uses "pixels" (px) for length measurement and "square pixels" (px^2) for area measurement. The "times" symbol "$\times$" – without a following unit – is used to jointly define the width and height of spatial entities, e.g., of a canvas with "20×10 pixels". Arithmetic operators and appropriate unit symbols are used for arithmetic calculations, e.g., $20px \cdot 10px = 200px^2$. Otherwise a "pixel" or a set of "pixels" refers to one or several pixel entities on the visualization canvas, e.g., the colored "pixels" of a line in a line chart.

Pixel Color. For each pixel of the pixel space of a 2D visualization, a color can be defined to determine the final appearance of the pixel in a rendered image.

Image Data. A set of pixels in combination with their pixel colors can be defined as a set of records with three or more attributes, denoted as image data.

Image data can have various forms, depending on the type of visualization. An image dataset may use one of the following common record types.

- Binary image data records (x, y, β) with $x \in \mathbb{N}_w$, $y \in \mathbb{N}_h$, and $\beta \in \{0, 1\}$, where x and y define the position of the pixel in the image and β defines whether a pixel is a foreground pixel or a background pixel. Foreground pixels are often colored black and background pixels are colored white.
- Grayscale image data records (x, y, γ) with x, y as above, and $\gamma \in \{0, 1, ..., 255\}$ defining the luminosity of a pixel.
- Colored image data records $(x, y, \rho, \gamma, \beta)$ with x, y as above and $\rho, \gamma, \beta \in \{0, 1, ..., 255\}$ defining the complementary *red*, *green*, and *blue* components of the color of a pixel.
- Colored image data records $(x, y, \rho, \gamma, \beta, \alpha)$ with x, y, ρ, γ, β as above and $\alpha \in \{0, 1, ..., 255\}$ defining the opacity of a pixel.

Other types of image data with alternative color models can likewise be defined as sets of records with one or more positional attributes and one or more color attributes.

When discussing the spatial properties of visualizations, this work mainly considers binary images and grayscale images of single data subsets, e.g., of a single line in a line chart. The corresponding geometric primitives in common data visualizations are usually rendered with one color for each data subset, i.e., with fixed values for ρ, γ, and β, varying only in the value for α; and only when considering anti-aliased rendering of geometries.

Technically the values for x and y can often be omitted, i.e., knowing w and h of an image, the image data can be represented as a sequence of $w \cdot h$ color tuples (β), (γ), (ρ, γ, β), or $(\rho, \gamma, \beta, \alpha)$. Vice versa, a binary image can be represented without specifying the actual color $\beta \in \{0, 1\}$ as a set of solely foreground pixels $(x, y) \in \mathbb{N}_w \times \mathbb{N}_h$. All other pixels are considered to be background pixel and must not be stored or transferred explicitly.

Note that many mobile devices and devices with high display resolutions often use physical, *device-independent pixels* (*dp*) to position elements on the screen. On such devices, when rendering geometric shapes, e.g., to the $320 \times 240dp$ of an HTML Canvas element, the browser internally uses the final, *native* pixel resolution, e.g., $640 \times 480px$, to maintain the image data for the final pixels on the screen. The remainder of this work aways considers this native resolution for visualization rendering and when discussing the properties of data visualizations.

2.4. The Visualization Pipeline

This work is related to systems and approaches that decompose the acquisition and subsequent visualization of the data as a modular process, considering different kinds of data processing operators. The following section lists and describes these widely-used approaches in context of the data aggregation approach of the present work. Thereby, analyses and models of the visualization process are provided by various sources in the literature, describing a variety of ways to set up a *visualization pipeline* [98, 48, 14, 80].

Visualization Pipeline. A visualization pipeline is a model for data processing in context of data visualizations. The model is used to define the consecutive data processing steps at various layers of an end-to-end visualization system. It fully defines how data entering the visualization pipeline is transformed to a visual representation.

In general, this pipeline comprises three high-level steps [48] for transforming raw data to a visual presentation on the screen, as depicted in Figure 2.4.

Figure 2.4.: Common three-step visualization pipeline model.

1. *Filtering* of data records to provide an interesting subset that should be visualized.
2. *Mapping* of data records to geometric primitives, whose coordinate values and attributes define position, size, shape, and color of a drawable object in the 2D space of the visualization.
3. *Rendering* of geometry data to discrete, colored pixels, i.e., to image data.

More comprehensive models also explicitly model the steps of data acquisition, such as importing files, and the composition of rendered images. For instance, Upson et. al [98], define such a pipeline in their *Application Visualization System* (AVS), depicted in Figure 2.5. They add a *source module* (a) for data acquisition and an *output module* (c) for post-rendering tasks.

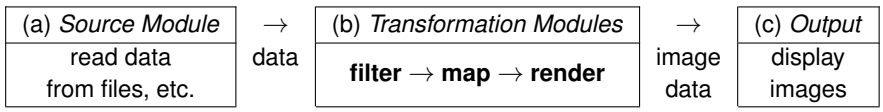


Figure 2.5.: Components of the AVS visualization pipeline.

The visualization-dependent data processing in AVS is modeled by configuring and connecting several *transformation modules* (b). Following the described three-step model, a combination of transformation modules eventually defines how to filter, map, and render data to pixels. The transformation modules support the following data operations.

- *data-to-data* operations, e.g., data filtering, scaling, and interpolation
- *data-to-geometry* operations, e.g., contour and surface generation
- *geometry-to-image* operations, i.e., geometry rendering
- *data-to-image* operations, i.e., image or volume rendering

There are even more detailed models of the visualization pipeline [25, 14], further decomposing the data transformation steps. For instance, Chi and Riedl [25] use a seven-step model to emphasize intermediate states of the original

three-step model. They define *value operators* to conduct data-to-any transformations and *view operators* for geometry-to-geometry, geometry-to-image, and image-to-image transformations. They model the intermediate results as reusable states, allowing for the definition of reusable data flows, e.g., for several views on the same data set. In their envisioned end-to-end visualization system that defines data visualizations using a sophisticated visualization pipeline model, data processing could be optimized jointly for the end-to-end data flow of all users. However, it also requires full control over all involved software components in the pipeline, and thereby presents a very different approach, compared to this work that aims to transparently integrate with existing systems.

Contribution

In regard to existing models of the visualization pipeline, this work adds the definition of a specific class of *data-to-data* operators for the task of data reduction. The operators are a complementary option for the first data-centric *Filtering* step in the three-step visualization pipeline (cf. Figure 2.4). They also simulate some aspects of the subsequent *Mapping* and of the *Rendering* steps, as illustrated in Figure 2.6.

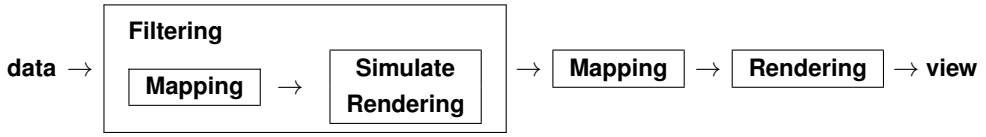


Figure 2.6.: VDDA as filtering step in the visualization pipeline.

Thereby, even though a visualization-driven data reduction requires additional knowledge about downstream visualizations, all developed operators are solely data-centric filtering operators, having raw data as input and producing a reduced set of raw data records as output, i.e., without changing the original schema and data format.

Moreover, the developed operators are not restricted to a specific type of visualization system or visualization pipeline model. If required, they can be used with any dataset or intermediate dataset between the data source and the first downstream image dataset or geometry dataset. As a result, the developed data reduction technique is fully compatible also with extended visualization pipeline models. For instance, in the described model of Chi and Riedl [25], the considered data reduction can be modeled as low-level *Data Stage Operator*.

Chi and Riedl also show that many kinds of apparently custom visualizations can be formalized using their model. Similarly, a visualization-driven data reduction can be developed for virtually any type of visualization, following the data reduction methodology used in this work. However, a detailed definition of specific VDDA operators for complex, domain-specific types of visualizations is out of scope of this work.

2.5. Visualization System Architectures

The main components of a visualization system, introduced in Section 2.1, can be deployed in different ways to implement a visualization pipeline. In fact, there is a plethora of architecture designs and real-world implementations of visualization pipelines. In context of a visualization-related data reduction, they can be classified into the following three main categories, complemented by a fourth category for the proposed system.

Monolithic systems (A)

Monolithic systems, such as most desktop applications (cf. Figure 1.1), can implement a complete visualization pipeline, with all components running directly on a desktop machine with potentially powerful hardware. In this architecture, large volumes of data are often imported to the application's internal data store, e.g., from CSV files, but also from an RDBMS, before the user is actually able to analyze and visualize the data. However, once the data is loaded, operations on the data can be performed quickly, since there is no limiting network connection in the visualization pipeline. Essentially, monolithic systems do not use any data reduction and require a complete apriori transfer of the underlying raw dataset.

Image-based distributed systems (B)

In many scenarios, small rendered images are transferred to the visualization client, instead of the actual data records, stored in an RDBMS or in a larger data warehouse. These pre-rendered images may be requested by a variety of applications, e.g., over HTTP using specific request parameters to identify a specific dataset and define the desired image resolution. In such systems, geometric transformation and rendering to pixels are conducted in the backend. The generated images often have a small fixed resolution to enforce data

to be delivered in a reduced format to the data consumers. Essentially, image-based systems restrict access to the data and compute and send images instead of data to visualization clients.

Data-driven distributed systems (C)

Many web applications work on large volumes of data that are stored in an RDBMS (cf. Figure 1.1). Thereby, the attached database often runs on a different host than the back-end logic of the web application that issues the visualization-related queries. In this common scenario, potentially large volumes of data have to be transferred between the database and other back-end components over the local network. With interactive visualizations in modern web applications implemented at the client-side, the acquired raw data will often be forwarded to the visualization client, where it is transformed to geometry data, e.g., using a JavaScript-based visualization framework, to be eventually rendered to screen pixels, e.g., using the rendering capabilities of a modern web browser. An additional data reduction component may be used in the back-end, implementing a numeric data compression [74] or computing time-based averages of the numerical data [67]. Essentially, data-driven systems provide access to raw or pre-aggregated data via ad-hoc queries. Large query results may be reduced subsequently by additional downstream components that conduct a data-driven, i.e., visualization-agnostic, data reduction.

Visualization-driven, data-centric system (D)

The aforementioned data-driven approaches consult only the generic properties of the raw data, such as data volume and value distribution. In contrast, this work provides a visualization-driven data reduction, incorporating the properties of the desired visualization, such as the width and the height. This allows the system to choose an appropriate data reduction for each considered visualization and to obtain the smallest possible subset that is required to effectively visualize the raw data. The developed solution is primarily *visualization-driven* but also aims to be *data-centric*, by defining the considered data reduction at the query-level, allowing for a reduction of large data volumes as early as possible in the visualization pipeline.

Table 2.1 compares the three described architectures to the visualization-driven and data-centric system developed in this work, listing their most important properties and estimating the achievable interactivity and the bandwidth requirements of each system. Table 2.1 also briefly illustrates how each

#	system architecture	data flow, transferred data, point of data reduction (⚙)	data reduction	inter-activity	band-width
A	monolithic	raw big data	no explicit reduction	++	−−
B	image-based	pixels	through rendering	−	+
C	data-driven	data	after query processing	+	+
D	vis.-driven + data-centric	data	as query-level aggregation	++	++

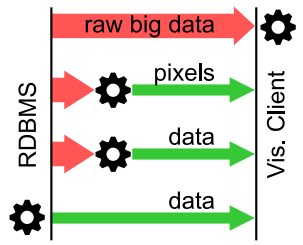


Table 2.1.: Comparison of visualization system architectures.

type of system reduces and transfers the acquired data. Thereby, a monolithic system A can provide a high interactivity as soon as all records are acquired from the database. However, regarding bandwidth requirements, a transfer of all records also constitutes the worst case scenario. Image-based systems B require much less network bandwidth and can reduce high data volumes to relatively small images. On the contrary, representing data as static images in systems B is the least interactive form of visualization. Improved interactivity is provided by data-driven systems C that are moreover effective in reducing the data, using generic sampling, aggregation, or compression methods. Theoretically, a data-driven system C may also be data-centric like system D and thereby lower the system-internal bandwidth requirements.[2] Nevertheless, by relying on generic data reduction methods, a purely data-driven system C will fail to obtain correct visualizations of the original data, suffering from missing details or even defective visualizations. This diminishes interactivity, since incomplete and defective visualizations require additional effort from the user to follow up on the perceived errors or missing details, i.e., configuring and issuing additional queries to the database. The proposed data-centric and visualization-driven system D provides the highest level of interactivity by serving a correct visualization for any given query. Moreover, system D can better utilize the available bandwidth by acquiring the minimal subset of the originally requested data that is required to render a correct visualization.

[2] For brevity, this work does not consider all kinds of combinations of architectural properties as specific system categories.

2.6. Existing Data Visualization Systems

The following section lists examples of existing visualization systems, categorizes them according to the architectures A to D defined in Section 2.5, and describes their relation to this work.

2.6.1. Visual Analytics Tools

Many visual analytics tools are systems of type A that do not apply any visualization-related data reduction, even though they often contain state-of-the-art data engines [104] that could be used for this purpose. For the present work, four common candidates for such tools were evaluated.

- Tableau Desktop 8.1 (tableausoftware.com)
- SAP Lumira 1.13 (saplumira.com)
- QlikView 11.20 (clickview.com)
- Datawatch Desktop 12.2 (datawatch.com)

A discussion of each tool is provided in Appendix A. Thereby, none of these tools was able to quickly and easily visualize a high-volume numerical dataset, having 1 million records or more. For all but Lumira, the conducted experiments resulted in memory issues or crashes, e.g., when trying to configure a simple line chart to display the records; the Lumira system only prevailed by rejecting to visualize any dataset that would contain more than 10000 records. In context of the developed visualization-driven data reduction, all tools support the acquisition of data from a relational database or may even provide a tool-internal data engine, working on a copy of the data. Each of these visual analytics tools could theoretically implement the proposed data reduction techniques, resulting in better support for visualizations of large datasets.

2.6.2. Big Data Visualization Systems

The data reduction techniques, presented in this work, are designed to work with any high-volume dataset. However, the developed techniques usually require the database to entirely process the visualized dataset at least once. In some scenarios, this may still result in very long query answer times, even for modern, in-memory database architectures [85]. The required *full scan* of the data can be avoided by materializing aggregated data in specialized

data visualizations systems, such as imMens [75]. Such systems do not rely on traditional database systems, but implement their own storage models and customized data processing techniques.

Another example is the ScalaR visualization system [16] that uses SciDB [28] to conduct highly parallelized data aggregation on the raw data, i.e., without the need for materialization of aggregates.

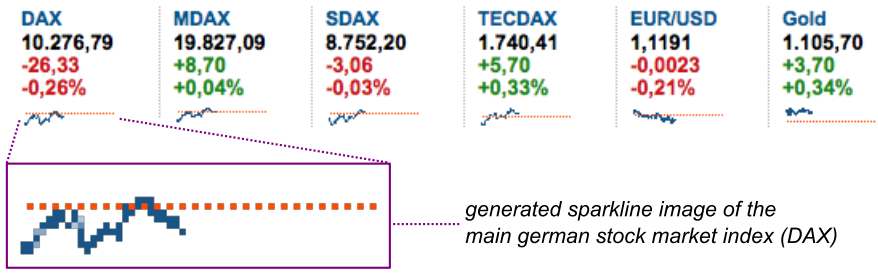
These exemplary systems are able to provide interactive response times for analytical queries on precomputed aggregates or even on the raw data. However, given that the original data is already securely stored in an underlying RDBMS, both examples must be considered as systems of type A. They require a full copy of the data, either to precompute aggregates (imMens) or to cluster and redundantly distribute the data in the system-internal storage layer (ScalaR).

The methods proposed in this work are complementary to the data management techniques described above. Thereby, materialization of aggregates, i.e., the redundant provision of reduced versions of the original data, can be combined with VDDA. For instance, in a first data-driven step, a system like imMens may select one of the materialized copies of the data for the user-requested data range. This initially allows the system to provide a reasonably-sized intermediate query result, independent of the chosen visualization. In a subsequent step, knowing the *type*, *width*, and *height* of the targeted visualization, the system can further reduce the intermediate query result using VDDA.

Similarly, VDDA can be implemented as a custom data aggregation in highly parallel systems like ScalaR. The visual aggregation is then conducted online on the raw data. Therefore, Section 3.6 will provide a detailed discussion on the parallelizability of the described data aggregation operators and describe how visualization-related queries can be decomposed into parallel operators running on partitioned datasets.

2.6.3. Common Web Applications

Image-based systems (type B) are used by many web applications, such as financial websites and websites of news agencies, to present market data using dynamically generated images. These images are provided with small fixed resolutions and present the data as sparklines (a) [97] or as complete line charts (b), with rendered labels and axis. Figure 2.7 illustrates these two ap-



(a) Excerpt of Yahoo Finance website using sparkline images of 60×16 pixels.

(b) Excerpt of Thomson Reuters website rendering complete charts on 320×300 pixels.

Figure 2.7.: Sparkline and line chart images on financial websites.

proaches, as found the Yahoo Finance website[3] and Thomson Reuters website[4] in September 2015.

Those systems reduce the data volumes by generating and caching raster images from the market data, and sending those instead of the actual data for most of their smaller visualizations. The benefit of such a system is to have full control over the generated image and – more importantly – to provide small and shareable query results to a multitude of users. The image generator of such an online visualization system can generate an image of a certain dataset once and subsequently serve the small resulting image file to millions of users, using proven, state-of-the-art content distribution systems [82]. However, purely image-based systems suffer from poor interactivity, since they do not provide

[3]Image acquired from https://de.finance.yahoo.com on 2015-09-10.
[4]Image acquired from http://reuters.com/finance/markets/europe on 2015-09-10.

the actual data records and thus do not allow for inspection of the visualized data, e.g., by clicking or touching a data point in the image.

While the present work does not consider generating images for the purpose of data reduction, the proposed visualization-driven data reduction has in common with image-based systems that both require knowing the type and additional parameters of the targeted data visualization. Formally, both approaches explicitly or implicitly aggregate the data in the $w \times h$ aggregation groups, i.e., pixels, of the visualization canvas.

For exploring data interactively, financial visualization systems of type B are often complemented with an interactive web application running in the browser. Such web applications are implemented as multi-tier client-server systems with data processing capabilities at several layers, often conducting a data reduction *between* the DBMS and the client [21, 41] and thus outside of the database. In such systems of type C, data may be reduced online for each incoming query, or data may again be pre-aggregated and materialized at several levels of granularity.

The web application may also solely rely on materialized data and restrict access to the raw data. In systems with amnesic storage models [44], materialized historical data may moreover not be available at all levels of granularity. For instance, at the Google Finance website, the user may view the data at several levels of granularity, as illustrated in Figure 2.8. Thereby, the DVS provides the most fine-grained data only for the last week, increasingly less fine-grained data for the last two weeks, the last month, the last year, and only very coarse-grained data since the emission of a share. In this particular system, small intervals of historical data cannot be visualized appropriately.[5] As states previously, the described system can be enhanced using the techniques developed in this work. Independent of the chosen storage model for the historical data, the pre-aggregated data can be reduced further using VDDA, before it is delivered to the visualization client.

2.6.4. Additional Systems

There are additional systems related to this work that are similarly data-centric or that similarly incorporate perceptual parameters.

[5] The presented analysis of http://google.com/finance?q=SAP was conducted on 2015-09-10.

Figure 2.8.: Provision of different levels of granularity.

Cross-Layer Systems

Similar to the presented visualization-driven, data-centric system, Wu et al. [105] describe Ermac, a data visualization management system (DVMS) that treats data visualizations as first class citizens to be optimized across layers. A data visualization is expressed declaratively in Ermac, using a declarative visualization specification language, similar to a declarative query language, supporting aggregation over visual *bins*. The definition of visual bins is similar to the considered grouping techniques of this work. Nevertheless, the proposed visualization-driven data reduction of the present work is complementary to a DVMS, providing visualization-specific data aggregations that can be implemented in Ermac queries.

Perception-Aware Systems

Additional perceptual properties, such as the color and the physical size of graphical objects, play an important role in how to effectively display large datasets. While previous research provides discussions and evaluation of such properties [103, 107], it is difficult to formally define a visualization perceptually and find a *simple* mathematical model for how the visualization is perceived by the human brain. In this regard, Pineo and Ware have shown that using a computational model of human vision allows for particularly well–perceivable visualizations [84], but their solution is non-trivial, requiring complex computations and repeated optimization. Conducting such operations on large data volumes causes a high processing overhead and delayed query results, and is out of scope of this work.

Proposing a much simpler approach, Chapter 6 will show how to include *spatial* and *perceptual* visualization properties in the data reduction process, for either improving the visual scalability of a chart matrix, or facilitating a precedent filtering operation that can further reduce the amount of acquired data.

2.7. Data Reduction

This section first discusses time series dimensionality reduction techniques, which are used to reduce large volumes of sensor data. Subsequently, the section discusses alternative and complementary data reduction techniques and lists additional related work.

2.7.1. Piecewise Data Reduction Techniques

The goal of most related data reduction techniques is, similar to the present goals, to obtain a much smaller representation of a complete time series. In many cases, this is accomplished by splitting the time series horizontally into equidistant or distribution-based time intervals and computing an aggregated value for each interval.

Piecewise Aggregate Approximation (PAA)

A commonly used approximation of a time series are mean values computed from subranges of the data. For instance, the piecewise aggregate approximation (PAA) [106, 67] approximates a large sequence of values $S = \{v_1, v_2, ..., v_n\}$ as a smaller sequence of $k = n/g$ mean values $S_{PAA} = \{a_1, a_2, ..., a_k\}$, computed for each g consecutive values of S, i.e., for each subsequence $S_i = \{v_{j+1}, v_{j+2}..., v_{j+g}\}$, with $j = g \cdot (i-1)$ and $i \in \{1, 2, ..., k\}$. Given a predefined window size g, and knowing the number n of records in the dataset, PAA provides an approximation of the original data of a predictable size k.

Adaptive Piecewise Constant Approximation (APCA)

A commonly used alternative to PAA is the adaptive piecewise constant approximation (APCA) [68] that introduces an additional error measure to ensure that the difference between the computed mean values and the underlying data does not exceed a certain threshold. This allows for using a dynamic window size, i.e., adding subsequent values to the current window and updating the current mean value, as long as the error stays within the defined threshold. The next window is then initialized with the previously violating value that caused the closing of the preceding window. APCA does not guarantee a certain data reduction rate, since it cannot compress sequences of highly variant values $v_i \in S$, i.e., resulting in windows of single values if the distance between any two consecutive values v_i and v_{i+1} is larger than the predefined error threshold.

Instead of computing a constant value for each window, other piecewise approximation techniques [40, 58] use lines, splines, or symbols from a limited alphabet to represent the original data. As described for APCA, these techniques often rely on an error measure to decide when to close a window or complete a line segment, e.g., as soon as the distance of the raw data values to the approximating line grows too large.

For data visualizations, existing piecewise data reduction approaches have the drawback of relying on approximating representations of the data, disallowing the reconstruction of original data records from the computed averages or compressed data formats. However, as this work will show, these records are often required to eventually derive a correct visualization of the original data. Visualizations of average values are perceptually very different from those of the original data, in particular when considering highly variant sensor signals.

2.7.2. Geometric Data Reduction Techniques

In addition to the aforementioned piecewise approaches, there are geometric approaches that aim to sample the data by determining the importance of each individual data point. The query result then includes only the important points and omits the unimportant ones.

Selection of Perceptually Important Points

Techniques for determining perceptually important points (PIP) are mainly used for time series dimensionality reduction (TSDR), which is required to speed up data mining tasks on large data volumes [40, 34]. Thereby, one approach for determining the importance of a point is to scan the data for major extrema [86, 39, 38]. This is similar to the proposed solution but fails to determine all important extrema that are required for a pixel-perfect visualization. An alternative approach is to measure the distance of each point to its preceding and consecutive points [43]. Thereby, different TSDR approaches consider different distance measures [42].

Distance Measures

The most common distance measure is the Euclidean (perpendicular) distance, as shown in Figure 2.9a, which is the shortest distance of a point p_j to a line segment $\overline{p_i p_k}$. Other commonly applied distance measures are the area of the triangle (p_i, p_j, p_k) (cf. Figure 2.9b) or the vertical distance of p_j to $\overline{p_i p_k}$ (cf. Figure 2.9c). The application of these measures for data reduction is not a

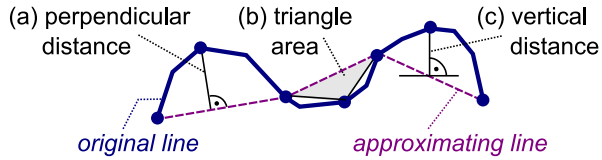


Figure 2.9.: Distance measures for line simplification.

novelty of TSDR techniques. In fact, they have been considered previously for line simplification [70, 91].

Line Simplification Algorithms

For the purpose of producing correct data visualizations, line simplification techniques are more appropriate than piecewise approaches (PAA, APCA) or simple applications of the described distances measures used by TSDR approaches [42]. Thereby, the over 40-year-old Ramer-Douglas-Peucker algorithm (RDP) [88, 30, 54] still provides superior results, rivaled only by the widely used Visvalingam-Whyatt algorithm (WisWy) [100]; depending on the use case. Line simplification algorithms are defined for arbitrary polylines and are not restricted to lines in a line chart, which are horizontally monotonic, i.e., with $t_i \leq t_{i+1}$ for any two consecutive records (t_i, v_i) and (t_{i+1}, v_{i+1}) of a time series $T = \{(t_1, v_1), ..., (t_n, v_n)\}$. Therefore, they can be considered as a generalization of TSDR approaches, since both rely on the same geometric distance measures. In contrast to the piecewise PAA and APCA, line simplification algorithms consider a time series in its entirety, instead of record by record. They iteratively process a given polyline, i.e., a set of connected straight line segments obtained from the original time series, to derive new polylines with fewer segments. Unfortunately, their iterative simplification process makes them more expensive to compute [52, 70, 91], in comparison to other related data reduction techniques that can often be computed in a single run over the data. Nevertheless, given the high perceptual similarity between the simplified and original lines, this work considers the line simplification techniques RDP and WisWy as a standard, regarding the achievable approximation quality for line charts. Section 4.8 provides a detailed comparison of the named techniques, also including the Reumann-Witkam line simplification algorithm (ReuWi) [89] that is – similarly to APCA and the developed VDDA, and in contrast to RDP or WisWy – computed sequentially in one run over the data.

2.7.3. Additional Data Reduction Techniques

Additional common data reduction methods and related techniques are the following.

Quantization techniques transform continuous data (real numbers) to discrete values (integers). Quantization is a lossy form of data reduction, since the projection of real numbers to discrete numbers $\mathbb{R} \rightarrow \mathbb{N}$ is a surjective function, i.e., the original set of real numbers cannot be determined from a quantized set of integer numbers. A visualization system may explicitly or implicitly reduce, i.e., quantize, continuous data to discrete values, e.g., by generating images, or simply by rounding real numbers, e.g., to have only two decimal places. A rounding function is also a surjective function and thus does not allow for a correct reproduction of the original data. For comparison with the proposed techniques, this work also considers rounding functions for data reduction, modeled as database query for a data-centric computation.

Data Sampling techniques [26] allow selecting small data subsets from large volumes of data to quickly provide approximations of the raw data. For instance, statistical databases like BlinkDB [13] or approximate aggregation techniques [18] can provide quick approximating answers to expensive queries. However, random samples and statistically approximating aggregates may not represent the best data subsets for the purpose of data visualization, as shown later in Section 4.8. Similar to aforementioned approaches, statistical databases may be extended using the proposed visualization-driven data reduction techniques. For instance, a statistical database may first provide a random sample of a large dataset, and subsequently improve the perceptibility of the sampled data by adding those records that are required to render a correct visualization on the $w \times h$ pixels of the targeted visualization.

Data Compression techniques, such as LZ77-based [108] packet compression, Huffman coding [57], or specialized compression of numerical data [74, 40] can be considered as additional *transport-level* data reduction technique. They are out of scope of this work, which only considers data reduction at the *application level*. Any subsequent, transport-level data reduction is complementary to VDDA.

Data Summaries of raw data, including simple, hierarchical [31], or amnesic [44] sums, counts, averages, variances, etc., can be very helpful for the user

to gain an overview of the data. Thereby, more detailed data summaries are provided for predefined groups of the data, e.g., for each data column of a database table and predefined temporal intervals. However, the predefined groups and intervals may not reflect the grouping of the data in the visualization, and the chosen summary values may not provide sufficient insight. For high-velocity and thus high-volume sensor data in particular, the actual shape of the underlying raw signal, e.g., perceivable as a line in a line chart, can be very useful to the user. An unaltered view on the raw data allows the user to intuitively scan the shape of the signal for patterns and anomalies and to quickly match the currently perceived shape with expected shapes from previous experiences. Summary and synopsis techniques are similar to the presented approach, in that computing data summaries usually relies on data aggregation at the query level, computed in the database. However, the goal of VDDA is to quickly provide an unaltered, virtually unaggregated view on the raw data.

Offline Aggregation is traditionally used in online analytical processing (OLAP) systems. An OLAP system aggregates the main data columns of a dataset and redundantly stores the aggregated values as condensed multi-dimensional data structure, the so-called OLAP cube. However, traditional aggregates of temporal business data in OLAP cubes are very coarse grained. The number of aggregation levels is limited, e.g., to years, months, and days, and the aggregation functions are limited, e.g., to *count*, *avg*, *sum*, *min*, and *max*. Similar to data summaries, pre-aggregated data may not provide the best representation for visualizing the raw data, e.g., high-volume time series data with a time resolution of a few milliseconds. As exemplified in Section 2.6.2, VDDA can be used as additional data aggregation before visualizing the pre-aggregated data. Moreover, it can be used as alternative data aggregation for precomputing the aggregates in visualization systems that rely on pre-aggregated data.

Online Aggregation techniques for data stream processing compute aggregated values from a set of incoming data records within a predefined time window. For online aggregation, the aggregation result for the current time window is continuously updated for each newly arriving record. However, online aggregation is formally still very similar to aggregation of static data, since the common aggregation functions *count*, *avg*, *sum*, *min*, and *max*, can be computed in single run over the static data and thus in a single run over a considered time window. In this regard, Liarou et al. [73] demonstrated how to

conduct efficient data stream processing on top of an existing general-purpose RDBMS. Consequently, all presented techniques in this work can be adopted also for online aggregation in traditional [12] and recent [23] data stream processing systems. Indeed, the need for interactive, real-time visualizations of high-velocity streaming data was one of the starting points for this work [63].

Content Adaptation techniques provide images and videos at different resolutions and compression ratios for web-based systems [76], e.g., to provide reasonably-sized content for consumers with large and small devices. Content adaptation is similar to the proposed approach, in that it is also driven by the requirements of the visualization. However, the presented approach is more generic, since it does not rely on a few static representations of the data, such as a set of multi-resolution images, but allows for dynamic queries to the database. Working with live data, a DVS using VDDA can extract the perfect data subset for any ad-hoc query and any individual visualization client, based on spatial and perceptual limits of the considered visualization.

Visualization-Driven Data Reduction incorporates the spatial properties of a visualization for data reduction. For instance, similar to the present approach, Burtini et al. [21] use the *width* and *height* of a visualization to define parameters for time series compression techniques. However, they describe a client-server system of type C (cf. Figure 2.1), applying the data reduction outside of the database. In the present work, all data processing is pushed down to the database. For line charts, they likewise consider w aggregation groups, but only select one aggregated value per group, whereby they are missing some of the important tuples required to draw a correct line chart, as discussed later in Chapter 4. There are other specialized systems that incorporate visualization properties as query parameters for the purpose of data aggregation. For instance, the ScalaR system [16] on top of SciDB [28], as already described in Section 2.6.2. However, existing works neither appropriately discuss the visualization-inherent data reduction at the pixel-level nor provide solutions for all common types of visualizations.

Visual Aggregation techniques aim to provide a better view on big data by summarizing thousands of records as a single mark in the visualization. Related techniques rely on custom visualizations that are not generally available in common data analytics tools [77]. Surveys on visual aggregation techniques in the literature are provided by Elmqvist and Fekete [33] and Shneiderman

[95]. In contrast to such highly specialized visual aggregation techniques, this work only considers *overplotting* as the unavoidable visual aggregation inherent in any data visualization. Overplotting occurs when rendering high-volume geometry data to discrete pixels. Essentially, the corresponding rendering algorithms procedurally implement the visual aggregation that the present work aims to formalize declaratively and thus more generally for all common chart types.

Even though this work considers visual aggregation through overplotting as the most ubiquitous and inevitable form of visual aggregation, it is clear that overplotting per se is not desirable and the DVS should provide additional measures to improve the visualization. Using VDDA, as described in Chapters 3, 4, and 5, at first only benefits the DVS by significantly reducing transferred data volumes without providing better perceivable visualizations. However, in Chapter 6, this work shows how to incorporate additional perceptual properties of the visualization in the aggregation process to further reduce the acquired data volumes and to perceptually improve the resulting visualization. Eventually, using the proposed techniques will in the first place allow many existing visualization tools to visualize big data in and unaltered way, i.e., providing a virtually unaggregated view on the raw data. It is still subject to the viewer of the visualization to subsequently and iteratively redefine which data to visualize, e.g., by defining filters, requesting summaries, or simply by zooming into a heavily overplotted area of the chart. In any case, the resulting query can be used as input for VDDA to ensure only the eventually visible data is transferred from the database.

2.8. Data Visualization

There exist dozens of ways to visualize numerical data, but only a few of them work well with large data volumes. Bar charts, pie charts, and many other simple chart types consume too much space per record, i.e., per rendered shape [32]. The most common charts, used for high-volume numerical data, are shown in Figure 2.10. These are *line charts* and *scatter plots*, where a single data point can be presented using only a few pixels. Regarding space efficiency, these two basic chart types are only surpassed by *space-filling* approaches [65].

Line charts and scatter plots, but also bar charts are basic components of many visualization systems. They are an integral part of general purpose visualization tools and are the basis for a variety of additional chart types. In this regard, Mackinlay et al. [77] already formalized a comprehensive list of

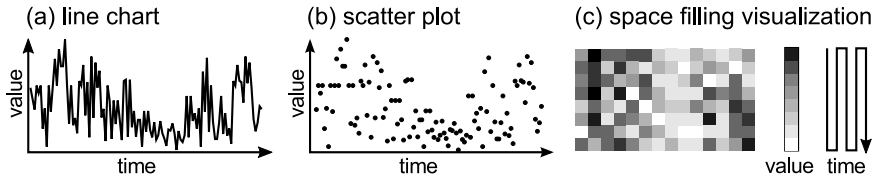


Figure 2.10.: Visualizations for high-volume datasets.

basic and composite chart types, to facilitate the development of an automatic chart selection system for Tableau.

2.8.1. Data Visualization Terminology

Partially borrowing from and extending Mackinlay's taxonomy, this work uses the following terms and concepts of data visualizations.

Mark Type. A mark presents one or more data records on the visualization canvas using a visualization-specific graphical object. Common mark types are *text*, *bars*, *lines*, and *shapes*, such as pixels, points, circles, rectangles, triangles, or crosses.

Note that, in contrast to Mackinlay's taxonomy, this work does not consider "Gantt" marks as an individual mark type, but as a specific form of *shape* marks.

(Basic) Chart. A chart is a visualization or sub-component of a composite visualization that presents one or more datasets on a single canvas. A chart jointly renders and potentially overplots data records in one shared pixel space.

This work extends Mackinlay's taxonomy primarily by analyzing and formalizing the *mark placement* in basic charts, the *series distribution* in chart matrices, and the *stacking* of single marks and entire subcharts.

Mark Placement. The mark placement of a chart defines how graphical marks are placed on the canvas of a single basic chart.

Thereby, marks in visualizations of categorical and ordinal numeric data are placed on the canvas according to a horizontal, vertical, or space-filling *sequential order.* Marks in visualizations of continuous numeric data are placed on the canvas according to one or more injective *projection functions* $\mathbb{R} \to \mathbb{R}$,

mapping value ranges of continuous data columns to the continuous layout dimensions of the 2D space of the visualization. Section 2.3 already formalized the linear projection functions f_x and f_y that this work considers being used by all common visualizations of continuous data.[6]

Series Distribution. The series distribution of a composite visualization defines how different datasets are distributed to the subcharts of the visualization.

To display multiple series in one composite visualization, this work considered the following two techniques.

Chart Stacking. A multidimensional dataset can be split up into several data subsets that are presented in individual subcharts. Subcharts can be stacked vertically, horizontally, or both in a chart matrix. Alternatively, the individual subcharts can be overlaid (z-stacking).

Vertical and horizontal stacking is a simple form handling visual overload in basic charts. Overlaying is helpful for saving screen space, but also for comparing high-volume data subsets. Chart stacking also benefits performance. If each subchart displays an entire data subset, not depending on the values of other subsets, then the creation of each subchart can be performed separately, e.g., by using separate queries to the DBMS and parallelized rendering of subcharts (cf. Section 3.6).

Mark Stacking. A multidimensional dataset can be split up into several data subsets that are presented using *correlated marks* in a single basic chart. The size and position of a mark in a correlation group influences the other marks in that group. Marks can be stacked vertically or horizontally.

Mark stacking requires the values of one data column to define the size of the marks. These values must be grouped by a second data column to assign each record to a specific correlation group and thus to a specific stack in the visualization. Compared to the stacking of separate charts, data aggregation and rendering of stacked marks must be conducted jointly on the stacked data subsets, as discussed later in Section 5.3.5.

[6] In practice, many DVS use *linear* projection functions as default but also allow for manually or automatically selecting a *logarithmic* projection function, i.e., a "log scale". A discussion of alternative projection functions is out of scope of this work.

category	*icon*	*chart type*	*marks*	*data*	*rank*
line charts		continuous lines	line	2Q	-
		discrete lines	line	1C + 1Q	4
bar charts		aligned bars	bar	1Q	2
		side-by-side bars	bar	1C + 1Q	-
		stacked bars	bar	2C + 1Q	3 (for > 3C)
		histogram	bar	1Q	-
		measure bars	bar	1C + 2Q	-
		space-filling vis.	bar[1]	1Q	-
2D plots		scatter plot	shape	2Q	5 (for 2Q)
		circle charts	shape	1C + 1Q	-
		Gantt charts	Gantt	1C + 2Q	6
2D grids		text table	text	1C/1Q	1
		heat map	bar[1]	1C + 1Q	-
		highlight table	bar + text	1C + 1Q	-
chart matrix		scatter matrix	shape	3Q	-

[1] If treated as special kind of bar chart.

Table 2.2.: Common visualizations in visual data analysis tools.

2.8.2. Considered Chart Types

Mackinlay's comprehensive collection of basic and composite chart types is listed in Table 2.2. This work moreover classifies these chart types into more general categories and also adds space-filling visualizations to the list. These are a less common but most space-efficient way to display large data volumes. As defined for Mackinlay's automatic chart selection system, Table 2.2 also depicts a representative chart icon[7] and lists the mark type, the supported minimal number of categorical (C) and quantitative (Q) data columns, and the rank that defines the priority at which a specific chart type is chosen. However, required only for automatic chart type selection, this rank is not considered for the remainder of the work.

The listed chart types constitute the reference list that the developed data reduction techniques need to cover. Consequently, this work considers the following basic and composite chart types.

[7] All chart icons are adapted vector-graphics versions of Mackinlay's original bitmap icons used in the paper [77].

Line Charts derive a point $(x, y)|x \in [0, w], y \in [0, h]$ for each record of a single series and connect each two consecutive points with a line segment to display each series as a connected line. The lines of different series may overlay each other. *Continuous line charts* display datasets with two continuous data columns projected to the continuous layout dimensions $x \in [0, w]$ and $y \in [0, h]$. *Discrete line charts* divide the canvas into $w' \leq w$ segments and horizontally center the projected endpoints (x, y) in the corresponding segment.

Bar Charts derive a bar with a height $h_{bar} \in [0, h]$ for each record of a single series. The bars are positioned sequentially along the horizontal layout dimension $x \in [0, w]$. The bars of different data subsets are not overlaying each other, i.e., either a bar chart is an *aligned bar chart* displaying a single series with up to w bars, or the bar chart is a *side-by-side bar chart* displaying up to $\lfloor w/n \rfloor$ bars of n series. Bar charts with vertical bars may conterminously be called *column charts* to distinguish them from *bar charts* with horizontal bars positioned along the vertical layout dimension $y \in [0, h]$. *Measure bars* can display one additional data column using the color of the bars. *Histograms* are used to display the distribution of the data, but are graphically not different from basic bar charts.

2D Plots derive a point $(x, y)|x \in [0, w], y \in [0, h]$ for each record of a single series and display graphical marks at each position (x, y) on the canvas. The size of the marks may be constant, e.g., using single pixels in *scatter plots*, or it may vary according to the values of another data column, e.g., using variable-sized circles in *bubble charts*. The marks of different series may overlay each other. *Circle charts* are 2D plots with a reduced horizontal resolution of $w' < w$ pixels, a fixed mark size, and the marks centered horizontally in the w' chart segments. *Gantt charts* are 2D plots with a reduced vertical resolution of $h' < h$ lanes, the rectangular marks centered vertically in each lane, and varying only the width of each mark, starting at position (x, y) and stretching to the right according to the values of a size-determining data column.

Space-Filling Visualizations use all pixels on the canvas to represent a sequence of up to $w \cdot h$ records of one single data subset. The order in which pixels are assigned to data records is usually non-linear and defined using a space-filling algorithm [65]. The most common space-filling algorithm is to render the pixels, for a list of up to $w \cdot h$ records, from top to bottom and from left to right, as illustrated in Figure 2.11a. However, naive fill techniques do

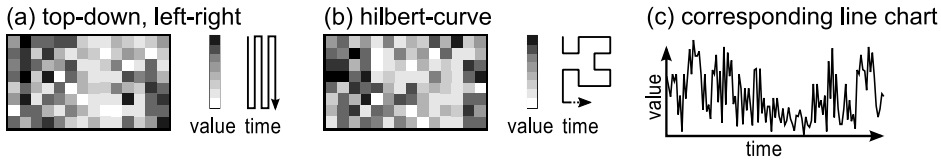


Figure 2.11.: Different fill patterns for space-filling visualizations.

not provide good data locality, i.e., with related data points being presented spatially close together. Another common fill pattern with better data locality is the Hilbert curve [55], illustrated in Figure 2.11b. For comparability, Figure 2.11c also depicts the visualized dataset as line chart.

2D Grids group the canvas into $u \times v$ grid cells, i.e., into u grid columns and v grid rows based on the groups defined by two ordinal data columns. Thereby, 2D grids display up to $n \leq u \cdot v$ individual data subsets, using one or more marks per cell to summarize the underlying data subset. For instance, each cell may display an average or maximum value, either as a string in a *text table*, as a shape of variable size or color in a *heat map*, or using a string, shape, and additional color in a *highlight table*.

Chart Matrices are 2D grids, where each cell represents the values of its data subrange as a basic chart instead of displaying only a summary of the data. Most forms of chart matrices, e.g., s*catter matrices*, *line chart matrice*s, *bar chart matrices*, and even space-filling matrices, can display one or several data subsets per subchart, according to the rules defined for the corresponding basic charts. In general, they can be considered as a mere composition of individual basic charts. The restrictions of the basic charts also hold a corresponding chart matrix. For instance, *aligned bar charts* in a chart matrix display at most one series per subchart and thus may require a large number of subcharts to display a high-dimensional data set. However, using an equal number of bars in each cell, they provide good comparability of the different series. In comparison, *side-by-side bar charts* in a chart matrix can display $\lfloor w/n \rfloor$ bars per subchart per series and thus allow for a higher number series to be visualized. They provide better comparability of smaller data subsets displayed jointly in one subchart.

Completeness of Chart Types

The considered set of supported chart types does not contain uncommon or infamous types of visualizations, such as pie charts or 3D bar charts. Nevertheless, this work assumes the list to be comprehensive and sufficient for most kinds of data, given that Mackinlay et al. defined their chart types based on the knowledge and experience provided by much-cited works on graphics and visualization design [17, 37, 97], and on their years of experience in developing a world-class software product like Tableau.

Moreover, even though this work focuses on the most common chart types, the developed data reductions techniques also work for custom-built, domain-specific visualizations, such as a multi-resolution visualizations of time series data [49], by decomposing the process of their creation into a sequence of steps in the visualization pipeline (cf. Sections 2.4 and 5.3.4).

2.8.3. Scalability of Visualizations

Different types of visualizations can display different quantities of data. The related scalability of a visualization was first discussed by Eick and Karr [32], analyzing spatial, perceptual, and interactive aspects of data visualizations.

> *"**Visual Scalability** is the capability of visualization tools to effectively display large data sets, in terms of either the number or the dimension of individual data elements."* [32]

However, instead of using this informal definition, this work will formally define the scalability of a visualization using spatial and perceptual *capacity measures*, defined by specific properties of the visualization and specific properties of human perception.

For instance, on a UHD screen with 3840×2160 pixels, a scatter plot can have a width of $w = 3840$ pixels and height of $h = 2160$ pixels and display up to $3840_{px} \cdot 2160_{px} = 8.29$ Million records, i.e., when using single pixels to mark each record, and when the underlying dataset contains at least one record in the corresponding data range of each pixel. However, since most datasets are not distributed homogeneously, all but the space-filling visualizations will be subject to a lot of white space and will display significantly less data than the theoretical maximum of 8.29 Million records; particularly when the chart uses large mark types, such as the bars of a bar chart.

Similar to the above calculation for the scatter plot, Chapters 3, 4, and 5 will first discuss VDDA in the context of only the spatial properties of a

visualization. Chapter 6 thereafter complements the discussion by describing in detail, how to incorporate perceptual properties and how to approximate the white-space consumption in overplotting-prone visualizations.

2.8.4. Data Visualization in Interactive Systems

A big data visualization must not only present all information in a perceivable and understandable way, but also provide additional means for interacting with the data [53]. Interactivity features allow the user to gain additional insight and to explore the data in many directions. Interactive systems usually follow the *visual information seeking mantra*.

> *"Overview first, zoom and filter, the details-on-demand"* [94]

Consequently, this work considered an interactive data visualization system to work as follows.

1. The user selects which dataset to view, and the system provides an *overview* of the data. Therefore, the system
   a) first issues an aggregating query to an attached database
   b) and subsequently displays the data in a common data visualization.
2. The user zooms into a data subset, and the system
   a) uses the previously acquired overview data to provide an approximation of the new data subset,
   b) displays the data subset using a common data visualization,
   c) issues another query to provide the missing details,
   d) and updates the visualization.

This exemplary interactive visualization process describes several properties and concepts of interactive systems, whose relation to the proposed data reduction techniques are as follows.

Use of common visualizations. General-purpose data visualizations tools aim to provide perceivable visualizations of all kinds of data. Therefore, these tools are relying on a common set of general-purpose data visualizations (cf. Table 2.2). VDDA is defined for all these common chart types and it is transparent to the visualization. Consequently, any general-purpose data visualization tool can use VDDA for all visualization-related data requests on an underlying high-volume data source.

Data on Demand. In this work, high-volume datasets are considered to be acquired on-demand and potentially transferred over a network. Consequently, when visualizing large data volumes, any data reduction that is fast to compute and does not impair the resulting visualization can be used to speed up the visualization process. In particular, any database query corresponding to an interaction with the visualization can be enhanced using VDDA to speed up the data acquisition and subsequent visualization processing.

Zoom and Pan. When zooming and panning through the data the user expects data to be visualized immediately. This requires the system to acquire a suitable amount of overview data upfront and subsequently acquire the missing details on demand. Both of these data acquisition operations are for the purpose of data visualization and can be improved using VDDA. Therefore, a pannable visualization may be considered to have a virtual size $p \cdot w \times q \cdot h$, with p and q defining how many times the user may shift the visualization by one full pixel width w or pixel height h of the visible canvas, without having the visualization system requesting new data.

A system that combines “zoom and pan” with ”data on demand” and that uses VDDA for all visualization-related queries will eventually allow the user to rapidly zoom and pan virtually through the raw data, while consuming only a fraction of the originally required bandwidth.

Summary

This chapter introduced the foundation for this work and provided an overview of existing techniques and related work in the areas of data reduction and data visualization. In particular, this chapter introduced the canvas of the visualization and the projection functions f_x and f_y that are generally used for determining coordinates in the 2D space of a visualization. The chapter concluded with the definition of the considered types of visualizations and described how VDDA will complement existing interactive visualizations systems.

3. Query Rewriting for Transparent Visualization-Driven Data Reduction

A data visualization system (DVS) must acquire additional knowledge about the data to appropriately determine if a query result can be transferred from the database over the network. If the dataset is too large, the original request must be altered to request less data.

This chapter first describes the core idea and defines the basic data reduction principle considered in this work. This is followed by a definition of the considered relational data model, the considered data, and a description of the expected visualization-related queries. Thereafter, the chapter defines a query rewriting system as the foundation for the developed visualization-driven data aggregation (VDDA), showing how to incorporate the spatial properties of the visualization in a visualization-related query. After following up with a discussion of the correctness of the developed spatial data aggregation and of data aggregation over multiple data subsets, the chapter finally concludes by defining how to subdivide visualizations and how to partition the underlying data to allow for parallel execution of visualization-related queries on large datasets.

3.1. Core Idea

Acquisition, processing, analysis, and visualization of big data pose challenges at all layers of a DVS. In particular, the creation of interactive big data visualizations requires an end-to-end view on visualization-related data processing. Hereof, the present work pays particular attention and is grounded on the principles of the following two data processing steps, conducted at the beginning and the end of the visualization pipeline.

Data-Centric Processing. Modern database systems are used for storing and managing big data. They provide fast, in-memory access to the raw data and thus allow for running analytical queries directly on the data, i.e., without having to transfer raw data to a downstream software component for subsequent data analysis.

Visualization Rendering. For a data visualization, all incoming data records are eventually projected to the pixels of an attached raster display. The rendering procedure is determined by the specific type of visualization. Given that the cardinality of the input data is larger than the number of resulting pixels, visualization rendering is a form of implicit data reduction at the pixel level.

The related research fields are well understood by researchers and practitioners alike, with a plethora of research on techniques for efficient data storage and fast data processing on modern hardware architectures [90], complemented with research on big data visualizations [95, 33] (cf. Chapter 2). However, existing works from both fields often remain short on how the big data is delivered to the corresponding visualization. Big data processing and big data visualization are considered as separate tasks, and the DVS is eventually not aware of the rendering process, e.g., of the rasterization of lines in a line chart [20, 24]. Existing systems throw away extensive potential savings by not respecting the detailed properties of big data visualizations. They essentially ignore that there is a duality of reduction and visualization of big data.

Duality of reduction and visualization of big data. *Interactive visualizations require raw big data to be aggregated or sampled, but sampling or aggregation cannot be effective without knowing the visualization.*

Respecting this duality, the present work aims to provide a systematic, end-to-end view on the creation of visualizations of large datasets obtained from relational databases. In particular, this work considers the effect of overplotting at the pixel level, to define a corresponding data aggregation at the query level.

3.1.1. Implicit Data Reduction through Overplotting

As already illustrated for a bar chart in Section 1.3.2, overplotting conducts an implicit data reduction during the process of data visualization. It occurs when projecting a large dataset to a two-dimensional pixel raster, having a discrete width w and a discrete height h. Thereby, the underlying dataset – no matter how large it may be – is reduced to a limited number of $w \cdot h$ screen pixels. Similar to the illustrated bar chart in Figure 1.2 on page 10, many visualizations have additional constraints that moreover prevent them from using every pixel available on the screen. For instance, scatter plots may often use only a small percentage of the pixels to represent the data, with the majority of pixels remaining uncolored, since they are not related to any record of the visualized data.

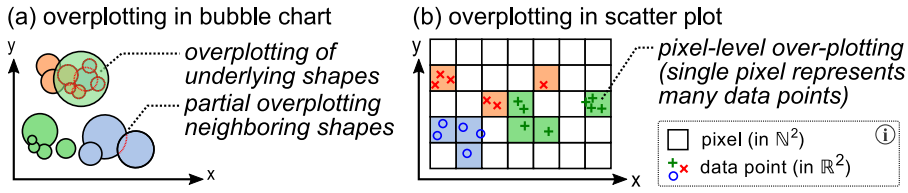


Figure 3.1.: Overplotting in (a) bubble charts and (b) scatter plots.

Furthermore, as illustrated in Figure 3.1, different types of visualizations are subject to different kinds of overplotting, depending on their corresponding rendering procedure. For instance, in a bubble chart (cf. Figure 3.1a), a larger circle may completely overplot, i.e., hide, smaller underlying circles. Independent of the circle size, any circle may also partially overplot other neighboring circles. In addition to this macro-level overplotting, the depicted scatter plot (cf. Figure 3.1b) shows that overplotting likewise occurs at the pixel level. In general, overplotting can be defined as follows.

Overplotting. A visualization of a dataset $\{(t, v) \in \mathbb{R}^2\}$ is subject to overplotting when two or more projected data points $(x_{real}, y_{real}) \in \mathbb{R}_w \times \mathbb{R}_h$ in the 2D space of the visualization are represented by the same pixel $(x_{int}, y_{int}) \in \mathbb{N}_w \times \mathbb{N}_h$.

Overplotting is prevalent in all types of visualizations, whether they display the data as small fixed-size pixels, large circles, or height-spanning bars. It is de facto inevitable if the acquired number of records exceeds the number of pixels on the screen.

The amount of overplotting, i.e., the ratio of drawn pixels to represented data records, also determines its data reduction potential. In Chapter 1, Table 1.1 on page 5 showed how $6M$ records of sensor data are reduced to the few thousand pixels in a common line chart. The potential savings related to overplotting are immense. A data reduction by two or more orders of magnitude is not uncommon for big data visualizations.

3.1.2. Visualization-Driven Data Aggregation

Overplotting is inevitable for very large and dense datasets and occurs in all types of visualizations. However, in previous works, detailed knowledge about the final visualization, i.e., about how data points are reduced to screen pixels,

has only partially been considered for reducing the cardinality of visualization-related queries [21, 16, 75]. A comprehensive definition of the data reduction principles inherent in each type of visualization has been missing.

The present work remedies this shortcoming, introducing a *visualization-driven data aggregation* (VDDA) that leverages the effects of overplotting in common types of data visualizations and simulates or approximates the overplotting process already at the query level. Therefore, VDDA uses common data aggregation functions, i.e., *min* and *max*, which are available in all major RDBMS, to define a grouping aggregation according to the pixel-level properties of the desired visualization. VDDA incorporates the visualization's pixel width w and height h to define separate aggregation groups. Each group is then approximated by its aggregated value. As a result, the cardinality of a VDDA query is inherently limited by the width and height of the visualization, i.e., to $w \cdot h$ or fewer records, depending on the type of visualization.

A more detailed discussion of VDDA for specific chart types of is provided later in Chapters 4 and 5. The remainder of this chapter first focuses on the general aspects of the developed solution, independent of the type of visualization.

3.2. Visualization Data Model

Many real-world data sources produce time series data, measuring and recording one or several values for a specific timestamp. In this regard, the present work mainly provides solutions for single time series and multidimensional time series, but also extends these solutions for arbitrary business data. Therefore, based on the data definition concepts for the relational algebra, this work considers the following time series data model.

Time Series. A time series is a binary relation $T(t, v)$, i.e., a set $T = \{(t_1, v_1), (t_2, v_2), ..., (t_n, v_n)\}$ of $n = |T|$ records. The relation T and thus each record $(t, v) \in T$ has two quantitative, i.e., numerical, data columns time t and value v, e.g., with $t, v \in \mathbb{R}$.

In terms of SQL databases, this work also denotes a relation $T(t, v)$ as *table* T with the data columns t and v. Multidimensional time series, i.e., time series with multiple data columns or sets of related time series are defined as follows.

Multidimensional Time Series. A time series of m measured dimensions is defined as a relation $T(t, a_1, a_2..., a_m)$, with unique timestamps $t \in \mathbb{R}$ and the recorded values $a \in \mathbb{R}$.

For small m, this work also denotes such a relation as $T(t, a, b, c, ..., m)$. The above model can be used to store data from multiple sensors in the data columns a_1 to a_m. However, considering that a single machine may easily have 500 different sensors [62], and that a manufacturing site may host thousands of machines, the number of data columns of T can be very large. An alternative representation for the considered multidimensional data is the following.

Multitude of Time Series. A multitude of time series is represented as a relation $T(id, t, v)$, with timestamps $t \in \mathbb{R}$, recorded values $v \in \mathbb{R}$, and an $id \in \{1, 2, ..., m\}$, defining which data source or series the record belongs to.

Adding new series to an existing multidimensional time series table $T(t, a_1, a_2, ..., a_m)$ in the database requires adding a new data column a_{m+1} to the table. This may not be desired for large numbers of series. The remainder of this work mainly considers the second model of multitudes of series in one table, where adding another series to the database is then achieved more easily by inserting data into the table, i.e., using a new unique series $id = m+1$.

If the number of correlated data dimensions is predefined, e.g., for a group of sensors in a single manufacturing machine [60], the two models can be combined. The grouped recordings can then be stored as multidimensional multitude of time series $T(id, t, a, b, ..., k)$, with k recorded values for each timestamp t for each group id.

3.2.1. Expected Query Load

Often, only a subset of the raw data should be visualized. Therefore, subsets of a multidimensional series and subranges within these subsets can be extracted from the data using the relational algebra.

For instance, given a relation $T(t, a, b)$ and knowing that t is the numerical timestamp and a and b are the numerical values of two different sensors, two separate time series relations are obtained by means of projection and renaming using the relational algebra expressions $T_a(t, v) = \pi_{t,v \leftarrow a}(T)$ and $T_b(t, v) = \pi_{t,v \leftarrow b}(T)$.

After the selection of one or more series, the visualized data can moreover be restricted to a specific time range, as illustrated in the following visualization-related query on a time series relation $T(id, t, v)$.[1]

[1] As already listed in the Notations of this document on page vii, selection subqueries are denoted as $\sigma_{<condition>}(<source>)$ and projection subqueries as $\pi_{<data\ columns>}(<source>)$.

$$\pi_{t,v}(\sigma_{t_1 \leq t \leq t_2 \wedge id=1}(T))$$

This query extracts the records of series $id = 1$ for a defined time range $[t_1, t_2]$. Most of the visualization-related queries, considered in this work, are such simple *select-and-project* queries, i.e., not incorporating potentially expensive join operators.

Eventually, a visualization may require to compare several sensor signals and thus issue a query over a multitude of series $T(id, t, v)$ as follows.

$$\pi_{id,t,v \leftarrow a}(\sigma_{t_1 \leq t \leq t_2 \wedge 1 \leq id \leq 100}(T))$$

This query request the data in the time range $t \in [t_1, t_2]$ for the series *ids* $\{1, 2, ..., 100\}$, leading to the following running Example 1 that is used frequently in the remainder of this work.

Example 1. A user chooses to visualize the last 10 minutes of a set of sensor signals as one or more line charts. In the visualization client, the user selects the *chart type* and the *time span* and defines a range of sensors to be displayed, e.g., the first 100 sensors, based on the order of their *id*. The visualization client uses these parameters to construct a visualization-related query as follows.

```
SELECT id,t,v FROM sensors
WHERE id <= 100 AND t >= $t1 AND t <= $t2
```

The measurements in the *sensors* table were recorded at $100Hz$. Consequently, the query yields $100Hz \cdot 100_{sensors} \cdot 10 \cdot 60s = 6M$ records, which are sent to the visualization client to be rendered as line chart. Given a smallest wire size of $3 \cdot 8byte$ per record, the query result comprises $144MB$, which takes over $11s$ to be transferred over a $100Mbit$ network. Encoded as JSON string, a subsequent client-side deserialization takes $3 - 5s$ and the rendering of $6M$ lines takes $10 - 50s$, depending on the user's browser and hardware[2]. As a result, the total post-processing time, without considering the query execution time is $24 - 66$ seconds.[3]

The above `sensors` table can be created in most SQL databases using the following exemplary data definition language (DDL) statement.

```
CREATE TABLE sensors (id int, t float, v float)
```

[2] Additional measurements and a simple online demonstration are available at http://openjuve.blogspot.de/2015/07/vdda-demo.html.

[3] Note that the deduced delays match with the response times measured in the evaluation of the state-of-the-art data analytics tools in Appendix A.

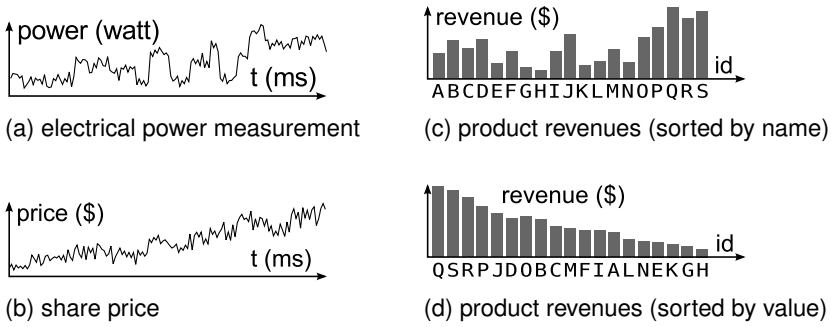


Figure 3.2.: Visualizations of ordered datasets.

To speed up access to each single series, a database index on the data column *id* can be created as follows.

```
CREATE INDEX sensors_id ON sensors(id)
```

An index on the series-identifying data column is considered as a prerequisite for multi-series tables in the remainder of this work. Knowing the *id* of a series, an index for table T on the data column *id* allows the query engine to quickly determine the location of the records of that series, e.g., starting at a certain offset in a file on the file system. Thereby, the index effectively avoids scanning the records of other series and thus will significantly speed up query processing when querying only small subsets of a multitude of series.

3.2.2. General Applicability of the Data Model

The considered time series model is applicable for other kinds of data, as described in the following. Any time series is ordered implicitly by the numerical values of its temporal data column. Time series are, for instance, the sensor measurements $T(t, v_{power})$ of a single electrical power sensor, illustrated in Figure 3.2a, or a list of records $T(t, v_{price})$ of a single share on the stock market, illustrated in Figure 3.2b.

For many data visualizations also categorical data can be considered as a *series*, even though such data may not have a canonical, e.g., temporal, order. For instance, a series of sales numbers $T_R(id, v_{revenue})$ of the products of a company requires an explicit order, defined by the visualization, i.e., either ordering T_R by *id*, as in Figure 3.2c, or by $v_{revenue}$, as in Figure 3.2d.

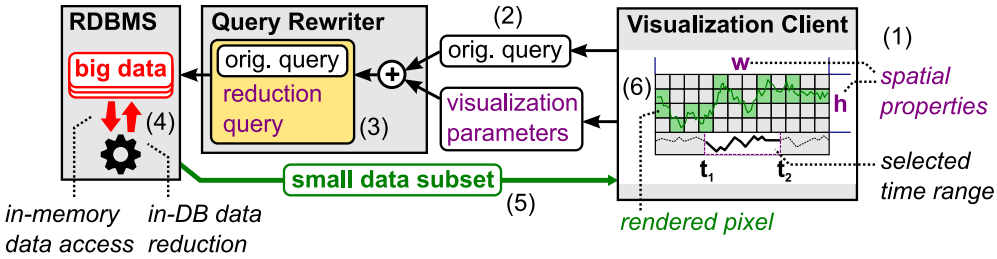


Figure 3.3.: Visualization system with query rewriter.

The above examples illustrate how to regard any kind of ordered data as *series*, i.e., a sequence of records to be rendered in a specific order by a corresponding data visualization. As a result and for brevity, the remainder of this work often restricts the discussion to simple time series $T(t, v)$ with one recorded value per timestamp. However, this simple model will frequently be complemented with a discussion of multidimensional time series, e.g, $T(t, a, b, c)$, and occasional digressions on additional types of relations, e.g., with categorical data columns. In this regard, Sections 5.4.1 and 5.4.2 will later provide complementary discussions and examples how to leverage the presented solutions also for categorical and ordinal numerical data in SQL databases.

3.3. Transparent Query Rewriting

The considered DVS architecture was described in detail in Section 2.1, with Figure 2.1 on page 13 depicting all relevant components and describing the considered data and control flows. This section now describes how visualization-related queries in such a DVS can be rewritten to conduct a visualization-driven data reduction on top of the original query result.

A rewriting can either be done directly by a high-level component, such as the visualization client, or by a low-level component, such as the query interface to the RDBMS. Independent of where the query is rewritten, the actual data reduction is always computed inside the database itself. Figure 3.3 illustrates the involved components, the rewriting steps, and the incorporated parameters of the considered query rewriting system. The steps to define and rewrite the queries, and to reduce and render the data are the following.

1. The definition of a query starts at the visualization client, where the user first selects a data source, a chart type, and the main query parameters.

The data source and its related parameters, e.g., the time range $[t_1, t_2]$, define the original query. This usually results in non-aggregating queries, such as the query for Example 1 on page 54.

2. In addition to the original query, the visualization client must expose the relevant spatial parameters, such as the width w and height h of the visualization canvas to the upstream components.

3. The query rewriter subsequently embeds the unaltered original query in a derived data reduction query. The data reduction operators are configured according to the received visualization parameters.

4. The derived data reduction query is issued to the database instead of the original query. The data reduction is computed by the database on top of the original query. Formulated as one integrated query, the database may take any measures to speed up query execution, without changing the semantics of the query.

5. The data-reduced query result is delivered to the visualization client, as an approximation of the raw data requested by the original query.

To not restrict the types of queries issued by the visualization client, the system specifically avoids altering the original query directly, i.e., it does not parse and analyze the query to subsequently augment any internal query parameters. The query is embedded as the first subquery of the rewritten query, with any additional data reduction subqueries building on top of it. As a result, the visualization client can define an arbitrary relational query, as long as the schema of the query result is compatible with the data model of the desired visualization.

3.3.1. Constructing the Rewritten Query

The goal of the rewriting is to apply an additional data reduction to those queries, whose result set exceeds a certain size limit. An exemplary *rewritten query* Q' is shown in Listing 3.1 and contains the following subqueries.

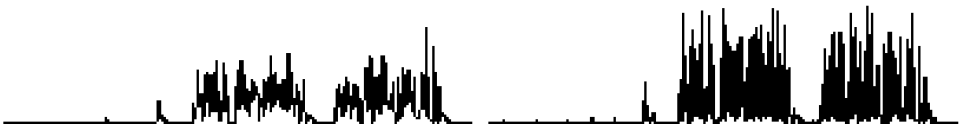
1. The original query Q,

2. a counting query Q_c on Q,

3. a data reduction query Q_r on Q,

Listing 3.1: Conditional visualization-driven PAA data aggregation.

```
WITH
Q   AS (SELECT t,v FROM sensors                    -- 1) original query
        WHERE id = 1 AND t >= $t1 AND t <= $t2),
Q_c AS (SELECT count(*) c FROM Q),                 -- 2) cardinality subquery
Q_r AS (SELECT min(t),avg(v) FROM Q                -- 3) data reduction subquery
        GROUP BY floor(                            -- computes aggregated values
           200 * (t - (SELECT min(t) FROM Q))      -- for w = 200 pixel columns
               / (SELECT max(t) - min(t) FROM Q)))
SELECT * FROM Q                                    -- 4a) use Q if cardinality
WHERE (SELECT c FROM Q_c) <= 10000                 --     is below limit
UNION
SELECT * FROM Q_r                                  -- 4b) use Q_r if cardinality
WHERE (SELECT c FROM Q_c)  > 10000                 --     is above limit
```



(a) resulting image (b) expected image

Figure 3.4.: Visualizations of original and PAA-reduced query result.

4. a cardinality check (conditional execution),
   a) to either use the result of Q directly, or
   b) to execute the data reduction query Q_r.

The SQL example in Listing 3.1 uses a width of $w = 200$ pixels, a predefined cardinality limit of 10000 records, and it applies a data reduction using a simple piecewise aggregate approximation (PAA) [67], as a generic measure for time series dimensionality reduction. A visualization of the original and rewritten query results are depicted in Figures 3.4a and 3.4b, illustrating that PAA cannot correctly reproduce the original visualization. Based on the given chart type, the query rewriter may use alternative, i.e., better-approximating, data reduction queries Q_r, as defined in Chapters 4 and 5.

From the targeted visualization, i.e., the line chart, the query rewriter considers only the width w to define the PAA parameters. In practice, the rewriter must moreover know which data columns correspond to which layout dimension, i.e., that the time t corresponds to the x-axis and the measured value v to the y-axis. In a DVS, this information may be acquired explicitly or implicitly.

The required mapping $\{t \rightarrow x, v \rightarrow y\}$ can be attached as explicit configuration to the original query by the visualization client, or it can be derived implicitly, e.g., from the order of the data columns in the original query. Alternatively, this information could be read from the metadata of the *sensors* table, which may provide additional semantic information about the data, e.g., defining t as an *analytical dimension*, and v as a corresponding *measure*. Predefining data columns as analytical *measures* and *dimensions* is a common approach in existing data analysis systems that operate on top of relational databases.

3.3.2. Visual Group Aggregation

A visualization-driven data aggregation requires a grouping function to group the data in the pixel space of the visualization canvas, and it subsequently uses an aggregation function to aggregate data in the grouped pixels, pixel columns, or pixel rows, i.e., in one of the components of the given pixel space, as introduced in Section 2.3 and shown in Figure 2.3 on page 20.

Grouping Function

The grouping requires aligning the value ranges $[t_{start}, t_{end}]$ and $[v_{min}, v_{max}]$ of the original data with the value range $[0, w]$ of the pixel columns and the value range $[0, h]$ of the pixel rows of the visualization canvas. Each pixel column or pixel row then corresponds to one of the w or h intervals of the value ranges of the visualized data.

Consequently, to compute a group key for each aggregation group, the query rewriter must know the visualization client's geometric transformation functions $f_x(t)$ and $f_y(v)$ (cf. Equation 2.1 in Section 2.3) and then *round* or *floor*, i.e., cut off, the resulting values in $[0, w]$ or $[0, h]$ to discrete group keys in $\{0, 1, ..., w\}$ or $\{0, 1, ..., h\}$. For instance, the line chart query in Listing 3.1 groups the data horizontally by time t, using the following grouping function $f_g(t)$, with $dt = (t_{end} - t_{start})$, and resulting in discrete group keys between 0 and $w = 200$.

$$f_g(t) = \lfloor w \cdot (t - t_{start})/dt \rfloor \tag{3.1}$$

The group keys of most 1D charts like bar charts and line charts can be computed using such a one-dimensional grouping function. Note that the maximum number of distinct group keys is $w + 1$ instead of w. However, Section 3.4 will show that the aggregated data in $w + 1$ groups always includes the aggregated data for the w pixel columns of the final visualization.

2D plots, require a vertical and horizontal grouping, incorporating both transformation functions f_x and f_y. For instance, a scatter plot uses the following grouping function $f_g(t, v)$, with $dt = (t_{end} - t_{start})$ and $dv = (v_{max} - v_{min})$, and resulting in group keys between 0 and $w \cdot h$.

$$\begin{aligned} f_g(t,v) \quad = \quad & h \quad \cdot \quad \lfloor w \cdot (t - t_{start})/dt \rfloor \\ & + \quad \lfloor h \cdot (v - v_{min})/dv \rfloor \end{aligned} \tag{3.2}$$

A more general definition of higher-dimensional grouping functions, including specific grouping functions for each considered chart type are defined later in Section 5.3.

Aggregation Function

The aggregation function derives aggregated values from each interval of the grouped data and is crucial for the resulting visualization. For the given example query in Listing 3.1, the resulting visualization of the data-reduced query result in Figure 3.4a differs significantly from the expected visualization of the original query in Figure 3.4b. The chosen aggregation function (PAA) computes average values and thereby significantly changes the perceivable shape of the time series, not matching the viewer's expectations. This example shows that the approximation quality of the visualization heavily depends on the type of aggregation function and that it must be chosen carefully. Chapter 4 will provide a detailed discussion on the utility of different types of aggregation functions for the purpose of visualization-driven data reduction specifically for line charts. Chapter 5 will extend the discussion to the other common chart types, and describe in general how to develop VDDA operators for any type of visualization.

3.3.3. Conditional Query Execution

Note that the presented query in Listing 3.1 defines a conditional execution using a union of the different subqueries Q and Q_r with contradictory predicates based on the cardinality limit. The conditional execution ensures that low volume data is not reduced unnecessarily.

In general, the described subqueries for reasoning on the cardinality are optional and per se not required for the data reduction. The decision to reduce the data or not can also be derived from first issuing a regular query that counts the records resulting from the original query. In practice, the DVS must ensure that the database can quickly execute both, the counting query

Q_c on the original query and the original query Q itself. Thereby, counting queries can usually be answered quickly by querying an index of the data instead of scanning the actual data records. Some original queries do to not directly return raw data as the query result, but produce new values, e.g., computing the $sum(v)$ per second for a range of sensors. For such queries, the DVS may decide to temporarily materialize the original query in the database, i.e., before counting the intermediate records and eventually computing the data reduction on top of the materialized result.

Using the conditional execution avoids the manual roundtrip to the database, since the original query Q, the counting subquery Q_c, and the data reduction query Q_r are integral parts of the entire rewritten query Q'. The query engine of the RDBMS can then find the optimal plan for executing the entire query efficiently. In particular, it may execute any user-defined query logic of the original query Q only once by leveraging that relational databases are able to reuse results of intermediate subqueries [59] and that the overhead of counting queries on intermediate results is negligibly small, compared to the execution time of the underlying subquery. The described features are present in most modern databases and the first facilitates the second. In practice, subquery reuse can be achieved via common table expression (CTE) [87], as defined by the SQL 1999 standard [79].

Essentially, the described query rewriting is a dynamic, data-centric approach. No intermediate data needs to be explicitly stored in the database or subsequently evaluated by the downstream components. No high-volume data is transferred downstream from the database.

3.4. Handling of Boundary Tuples

The grouping functions (cf. Equations 3.1 and 3.2) first project the data to the continuous 2D space of the visualization and subsequently cut off the fractional digits to derive discrete group keys. Thereby, rounding and truncation functions may result in additional groups, that are not part of the pixel space of a visualization. This is detailed in Figure 3.5, where several records within a time range $[t_{min} = 0.0, t_{max} = 5.0]$ are distributed to $w+1$ pixels. The considered visualization only has $w = 5$ pixels available. All data, including the first record at $t = 0.0$ can be assigned to a pixel, i.e., a corresponding aggregation group, except for the single last tuple at $t = 5.0$. The last record or a set of last records at $t = t_{max}$ projects to an additional aggregation group $w + 1$. A corresponding pixel may not be part of the visualization.

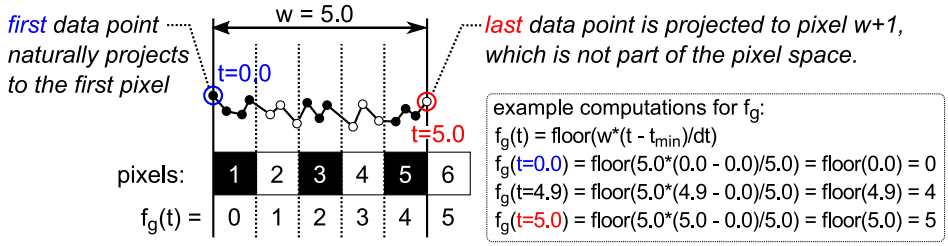


Figure 3.5.: Behavior of boundary tuples for pixel-based grouping.

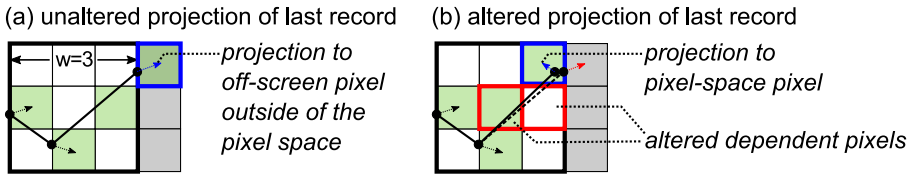


Figure 3.6.: Altered and unaltered projection of boundary records.

However, to ensure that all data is visible to the viewer, the visualization framework or the renderer of a DVS can define additional rules to handle such boundary records. For instance, they may shift the visual representation of the off-screen data slightly by a small sub-pixel distance, without changing the associated values of the shifted records. This may slightly alter the pixel color of pixels in the last and preceding pixel columns of a visualization, as illustrated in Figures 3.6a and 3.6b. Formally, these two data projection models are defined as follows.

Definition 1. An **unaltered projection** of data records to pixels determines for each record (t, v) a discrete pixel (x, y) using the discretized result of the projection functions f_x and f_y (cf. Equations 2.1), i.e., $(x, y) = (\lfloor f_x(t) \rfloor, \lfloor f_y(v) \rfloor)$.

Definition 2. An **altered projection** of data records to pixels uses the discretized result of the projection functions f_x and f_y to determine a discrete pixel (x, y) only for the data records (t, v) with $f_x(t) < w$ and $f_y(v) < h$. For any record with $f_x(t) = w$ or $f_y(v) = h$, the altered projection uses $x = w - 1$ or $y = h - 1$ to define the corresponding pixel.

The information if and how the DVS uses an altered projection of boundary records is additional knowledge that is theoretically required by a visualization-driven data reduction technique like VDDA. However, the visually reduced

dataset may alternatively provide correctly aggregated values for both cases. The aggregated value for the last pixel column w and the aggregated value for the off-screen pixel column $w+1$ must suffice to determine a corresponding aggregated value for a correct altered projection of the data. The same holds for the aggregated values for pixel rows h and $h+1$, in case the visualization uses a two-dimensional grouping.

The following reasoning defines for a reduced dataset, computed using the common aggregation functions, if and how they fulfill the described property of *boundary safety*, under the assumption that the $w+1$ aggregation groups are defined using the VDDA grouping function f_g (cf. Equations 3.1 and 3.2).

Lemma 1. *A reduced dataset is boundary-safe if it contains or allows to determine both (i) the aggregated records for an unaltered projection to $w+1$ pixel columns or $h+1$ pixel rows and (ii) the aggregated records for an altered projection to w pixel columns or h pixel rows.*

Note that an aggregation function only aggregates the values of one data column. For a two-column relation $T(t,v)$, the aggregation can be computed separately for the projections $\pi_t(T)$ and $\pi_v(T)$. Consequently, the subsequent reasoning can be restricted to single-column datasets, e.g., to $T_t = \pi_t(T)$.

Eventually, one has to show how any considered aggregation function allows combining the originally aggregated values t_w and t_{w+1} of the reduced dataset $A_{w+1} = \{t_1, t_2, ..., t_w, t_{w+1}\}$ to derive a new aggregated value that is equal to an aggregated value t_w^* that could have been computed from the combined set of values in the aggregation groups T_w and T_{w+1} of the raw data T_t. This aggregated value t_w^* replaces t_w and t_{w+1} in A_{w+1} to obtain the altered reduced dataset $A_w = \{t_1, t_2, ..., t_{w-1}, t_w^*\}$, required for a correct altered projection of the data. In other words, two aggregated values need to be combined to define a new aggregated value under the same aggregation semantics and this essentially requires the aggregation function to be associative.

Theorem 1. *The VDDA grouping function $f_g(t)$ provides a boundary-safe grouping for any associative aggregation function $f(T_t)$, i.e., with $f(\{t_a, f(\{t_b, t_c\})\}) = f(\{f(\{t_a, t_b\}), t_c\}) = f(\{t_a, t_b, t_c\})$ for any $t_a, t_b, t_c \in T_t$.*

Proof. Suppose a set $A_{w+1} = \{t_1, t_2, ..., t_w, t_{w+1}\}$ is *boundary-safe*. Suppose that $t_w = f(T_w)$ is the aggregated value of the w^{th} group T_w of a dataset $T_t = \pi_t(T)$. Suppose that $t_{w+1} = f(T_{w+1})$ is the aggregated value of the $(w+1)^{\text{th}}$ group T_{w+1} of T_t. Suppose there exists an alternative set $A_w = \{t_1, t_2, ..., t_{w-1}, t_w^*\}$ with $t_w^* = f(T_w^*)$. Now given an associative function

aggregation function	*associative*	*alternative*	*example (denoted as n-ary function $f(t_1, t_2, ..., t_n)$ on dataset $\{5\} \cup \{2, 3\}$)*
min	yes	-	$min(5, min(2, 3)) = min(5, 2, 3) = 2$
max	yes	-	$max(5, max(2, 3)) = max(5, 2, 3) = 5$
sum	yes	-	$sum(5, sum(2, 3)) = sum(5, 2, 3) = 10$
$count$	(yes)	sum of $counts$	$sum(count(5), count(2, 3)) = count(5, 3, 2) = 3$
avg	(yes)	$avg = \frac{sum}{count}$	$\frac{sum(5,sum(2,3))}{sum(count(5),count(2,3))} = avg(5, 2, 3) = 3.\overline{3}$
$median$	no	-	

Table 3.1.: Associativity of aggregation functions.

$f(\{t|t \in \mathbb{R}\})$ and given $T_w^* = T_w \cup T_{w+1}$, then $f(T_w^*) = f(\{f(T_w), f(T_{w+1})\})$. But then $t_w^* = f(T_w^*) = f(\{t_w, t_{w+1}\})$, since $t_w = f(T_w)$ and $t_{w+1} = f(T_{w+1})$. Consequently, A_w can be derived from A_{w+1}, and therefore, given that the grouping for A_{w+1} was defined using a grouping function $f_g(t)$, f_g is *boundary-safe* for the aggregation function f. □

Table 3.1 defines for the common aggregation functions, if they are associative or not and exemplifies, if possible, how to preserve their associativity using an additional or alternative aggregation on the same dataset. Thereby, the aggregation functions min, max, and sum are fully associative. The $count$ aggregation function is quasi-associative, given that original counts are *added* and not *counted*. The avg aggregation function is quasi-associative, given that it can be defined as $avg(T) = sum(T)/count(T)$. Only the $median$ must be computed on the raw data and cannot be safely obtained from the computed medians of subsets of the data.

This concludes the discussion of the boundary safety of the VDDA grouping function (cf. Equation 3.1), in the context of the common aggregation functions used in database systems. Consequently, all common aggregation functions, excluding the $median$ function, can be used for a visual aggregation at the query level.

Note that the remainder of this work does not explicitly distinguish between a grouping of the data by w or $w + 1$, or by h or $h + 1$ groups. For brevity, and having shown that the considered grouping function provides correct group keys for either case, the remainder of this work focuses on discussing the case of grouping the data visually correctly into w or h groups.

Listing 3.2: Conditional data reduction query for multiple series.

```
WITH
Q   AS (SELECT id,t,v FROM sensors                  -- 1) original query
        WHERE id <= 100 AND t >= $t1 AND t <= $t2),
Q_c AS (SELECT count(*)/count(distinct id) c        -- 2) cardinality subquery
        FROM Q),
Q_r AS (SELECT id,min(t),avg(v) FROM Q              -- 3) data reduction query
        GROUP BY id, floor(                         -- aggregates over series id
            200 * (t - (SELECT min(t) FROM Q))      -- and pixel columns
                / (SELECT max(t) - min(t) FROM Q)))
SELECT * FROM Q
WHERE (SELECT c FROM Q_c) <= 10000                 -- 4a) use Q if cardinality
UNION                                              --     is below limit
SELECT * FROM Q_r                                  -- 4b) use Q_r if cardinality
WHERE (SELECT c FROM Q_c) > 10000                  --     is above limit
```

3.5. Handling of Multiple Series

To illustrate the basic structure of a rewritten query for a single series, Listing 3.1 used a simplified original query Q, selecting only one series with $id = 1$, instead of e.g., 100 series with $id \leq 100$, as required for Example 1. Nevertheless, the considered conditional data reduction queries for multiple series are very similar to those of single series. Indeed, it requires only minor modifications to the single-series query (cf. Listing 3.1 on page 58) to rewrite the original query of Example 1, and thus define a corresponding data reduction for all 100 series.

Listing 3.2 defines this query and the highlighted differences to Listing 3.1 are the following.

- The original query Q now selects multiple series and includes the series id as projected data column (cf. Example 1)
- The cardinality subquery Q_c computes the average number of records per series instead of the number of records of a single series.
- The data reduction subquery Q_r computes aggregated records not only for each pixel column, but also for each series id.

As a consequence, the query rewriter needs to be aware that the categorical data column id identifies unique series in the `sensors` table and that it is used by the DVS to distinguish different data subsets in the final visualization, e.g.,

using different line colors or line styles. For data visualizations like line charts, where individual series are rendered separately, the query rewriter always depends on the definition, incorporation, and acquisition of such an additional categorical data column. Otherwise, the DVS would not be able to compute a reasonable cardinality limit. Not grouping the data records by their *id*, the visual aggregation would consider all records to belong to a single series, computing combined aggregates per pixel column, and resulting in an erroneous visualization.

3.6. Parallelism

An important question for the management of big data is if and how the expected queries can be parallelized. Therefore, the following section first describes the parallelizability of the rewritten query and secondly describes general options for parallelizing visualization-related queries.

3.6.1. Parallelized Rewritten Query

The multi-series query in Listing 3.2 computes an individual visual aggregation for each selected sensor. Given that the RDBMS allows for data parallelism, e.g., by storing the data partitioned by *id*, the RDBMS can parallelize the data aggregation subquery Q_r. Therefore, the system may run 100 group key computations and 100 subsequent *min* and *avg* aggregation operators in parallel. The data can be further partitioned into separate time ranges, resulting in separately stored and separately processable datasets not only for each sensor, but also for each for each month, week, or day. For recordings of high-velocity sensor data, e.g., the velocity of a soccer ball at $> 1500Hz$ [81], the partition interval may be even smaller, e.g., comprising $1500_{Hz} \cdot 60_{sec} = 90k$ records in a partition for each minute of the soccer game.

As illustrated in Figure 3.7, the rewritten multi-series query in Listing 3.2 can theoretically be computed on the aforementioned partitioned dataset, in particular when considering the following properties of the used subqueries and operators.

1. The original query Q merely acquires and filters raw data records from the stored data, before passing it to the subsequent subqueries (cf. op_1 and op_2 of original query Q in Figure 3.7 on the next page). The filtering can be computed in parallel for each individual record in each partition of the raw data. The partitioned query result $Q^{part} = \bigcup_{i=1}^{n} Q_i$ then contains

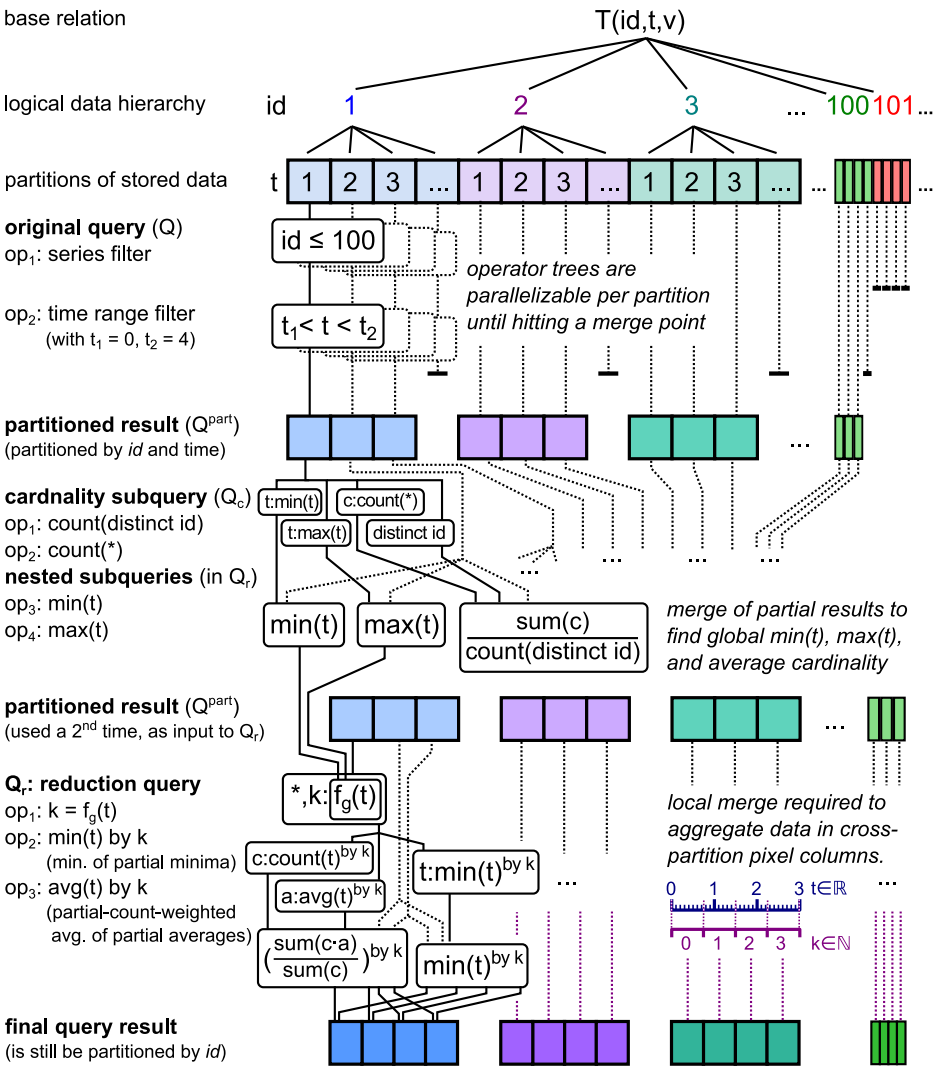


Figure 3.7.: Parallel processing of a multi-series data reduction query.

the result of the original query for each partition of the raw data. For some partitions this partial result is empty and further processing on this partition can be stopped.

2. The remaining subqueries require merging data from different partitions, with the following two consequences.
   a) The final subquery result can only be finalized after all computing nodes have produced a result for all input partitions.
   b) The partial results need to be transferred from the computation nodes to a central node that conducts the merge.

   However, all subqueries in Listing 3.2 produce very small partial results, e.g., one record per partition, resulting in low network bandwidth requirements for a subsequent *merging of small partials* (MSP).

3. The cardinality query Q_c can be computed by first computing intermediate results on each partial result Q_i, i.e., the cardinality c_i and the set I_i of distinct *id*s of each partition. All sets I_i and counts c_i can be combined to define the initially requested average cardinality of a series, i.e., using the MSP aggregation `sum(c)/count(distinct id)` over all intermediate results.

4. The nested boundary subqueries of Q_r can be computed by first acquiring the minima and maxima of the partial results of Q^{part} and subsequently computing the minimum of partial minima or maximum of partial maxima, which are again inexpensive MSP aggregations.

5. The data reduction query Q_r can be computed on Q^{part} by first computing a group key $k = f_g(t)$ for each record individually, and subsequently, computing the grouping aggregations $t = min(t)$, $a = avg(v)$, and a group *count* c for each partition Q_i grouped by k. As final step, the results of one or more neighboring partitions need to be merged to derive valid aggregated values for each series and for each pixel column, using an MSP aggregation `sum(c*a)/sum(c) GROUP BY k`. Note that, since the data is already partitioned by *id*, the originally required grouping by *id* is already ensured and the involved *min*, *avg*, *count*, and *sum* operations can be safely computed separately on the partitioned series.

6. The final query result $Q_r^{part} = \bigcup_{id=1}^{m} \bigcup_{k=0}^{w} (id, t_{min}^k, v_{avg}^k)$ is sill partitioned by the $m = 100$ *id*s, even though the grouping into $w = 200$ groups may

be different from the original range partitioning of the raw data of each series.

In general, a visualization related query can be parallelized if the data can be partitioned into independent subsets and if the queries can be decomposed to compute the results on each subset independently. Since the present query rewriting technique is designed to work with arbitrary original queries the parallelizability of the entire rewritten query Q' primarily depends on the parallelizability of the original query Q. In practice, most data visualizations, in particular of high-volume sensor data, will issue original queries similar to the query of Example 1, i.e., requesting several raw data columns from a database table and filtering the data using one or more categorical or numerical range predicates. These filter operations can be computed on each individual record and thus on each partition separately.

3.6.2. Parallelism in Data Visualization Systems

Most visualization-related queries can be decomposed into a set of independent queries that each request a fraction of the data. Therefore, the DVS may split the visualization into several separate smaller visualizations, as illustrated in Figure 3.8.

For instance, Figure 3.8a shows a stacked bar chart of two series. Assuming that the visualized range $[t_1, t_2]$ contains data from three days d_1, d_2, and d_3, the entire visualization can be considered as a composition of three separate visualizations, one for each day. The visualization client may then issue three separate queries to acquire the visualized data. The queries may each be executed on a different processing node, given that the underlying data is partitioned by day and distributed to one or more storage and processing nodes in the database.

The decomposition of the visualization and thus of the query can moreover be done transparently to the visualization, i.e., the visualization client may issue queries, without specifying how to partition these queries. The DVS automatically defines the partitioned queries, depending on the predefined partitioning of the data.

Independent of whether and how the data are partitioned, the DVS may still decompose the visualization and thus the query, to provide query results iteratively, e.g., when expecting the queries to be expensive to compute. For instance, for the scatter plot in Figure 3.8b, the DVS may issue four separate queries that are each faster to compute than one single overall query. Thereby,

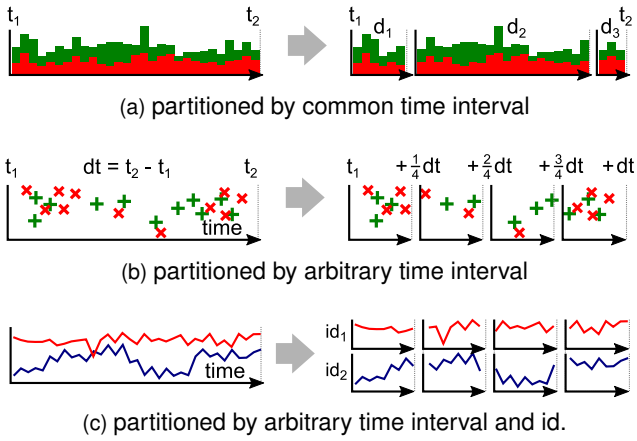


Figure 3.8.: Partitioning of time series visualizations.

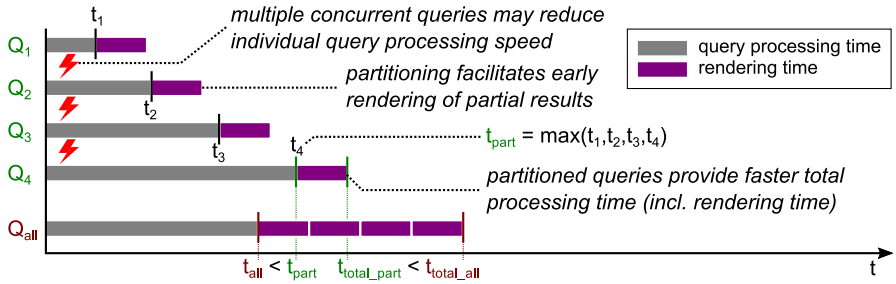


Figure 3.9.: Processing of partitioned queries.

the scatter plot can be filled with data iteratively, providing the user with early partial query results. This form of parallelism may also improve the total time until the visualization is visible, even if the four separate queries together are overall not faster to compute than a single query, but particularly if the acquired large data volumes are causing high rendering costs, as illustrated in Figure 3.9.

Partitions can be defined for various data columns and can generally be leveraged to decompose a visualization-related query. For instance, Figure 3.8c shows a line chart of two series in the same time range. Thereby, each series can be handled separately, as described in Section 3.5, and may moreover be acquired using separate queries for consecutive time ranges. The derived

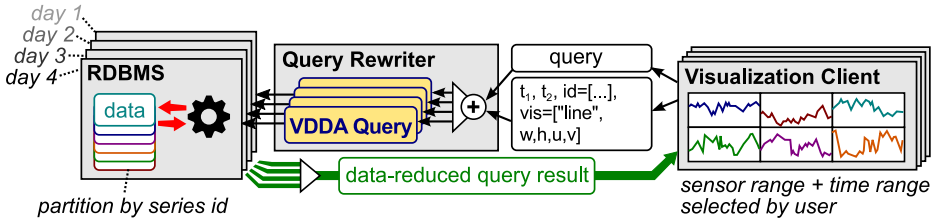


Figure 3.10.: Parallelization in the DVS architecture.

queries are then processed on the corresponding partitions, computing a separate visual aggregation for the specific subchart.

In Figure 3.10, the depicted DVS automatically decomposes incoming original queries, rewriting them to a set of data reduction queries, conducting a chart-specific aggregation on the partitions of the data. The visualization client still issues only one original query, including the visualization parameters, e.g., the width w and height h of the *line* charts of a chart matrix with u matrix columns and v matrix cells. In this system, the client explicitly specifies the requested data ranges, i.e., adding the selected time range $[t_1, t_2]$ and set of series *id*s as visualization-related parameters. Given that the query rewriter knows the distribution of the partitions, it can then define all separate partitioned queries, without requiring to analyze the original query and without a precedent probing and acquisition of metadata from the database.

This concludes the discussion of the parallelization options in data visualization systems. In practice, partitioned execution of queries should not be the responsibility of the higher-level components of a DVS, such as the query rewriter. This task should be done by the query optimizer of an RDBMS, using its integrated query analysis and optimization algorithms for detecting and leveraging the potential partitioning options.

Summary

This chapter elaborated on the idea of leveraging overplotting for the purpose of data reduction. Therefore, this chapter introduced the concept of visualization-driven query rewriting, describing how to incorporate the most important visualization parameters in defining a relational query that conducts a data reduction inside the database. The chapter moreover discussed the properties of these queries, i.e., the formal correctness of the considered

grouping function, the handling of queries on multiple series, and the general parallelizability of the queries on multitudes of partitioned time series.

4. M4: Visual Aggregation for Line Charts

The goal of the described query rewriting system is to conduct a data reduction very close to the data, i.e., at the query level in systems with relational databases. The chosen data reduction and its parameters are defined by the spatial properties of the targeted visualization. Therefore, the considered data reduction must simulate or approximate the visualization rendering process, i.e., the geometry rasterization procedure, using common data aggregation operators, such as *min*, *max*, or *avg*. However, particularly for line charts, the literature does not provide a comprehensive discussion of how data aggregation is related to the rasterization of geometries. This chapter provides this missing discussion. Therefore, it first describes several classes of data aggregation operators, discussing their utility for line charts, and leading to the definition of the M4 aggregation that precisely models the line rendering process. Thereafter, the chapter proves the correctness of this M4 aggregation and discusses its complexity. After a discussion of complementary and alternative implementations for the described data aggregation, the chapter concludes with a comparison of M4 with existing approaches.

4.1. Line Charts

Line charts are the most commonly used type of visualization for displaying continuous signals. For comparison, Figure 4.1 illustrates the visualization of a continuous sensor signal, using several common chart types.

In the depicted line chart (a), one can observe how drawing a line between each two consecutive points visually indicates the continuity of the signal to the viewer. This is not the case when rendering the points separately using disconnected marks in the depicted scatter plot (b). The depicted bar chart (d) is better in this regard, but overemphasizes peaks in the signal. A scatter plot and a line chart can be easily combined (c) to emphasize both, the individual points and the continuity. Figure 4.1 moreover depicts a table (e), which is usually used as a complementary view on the data, providing an easy way to read individual and neighboring values precisely. The last example is a

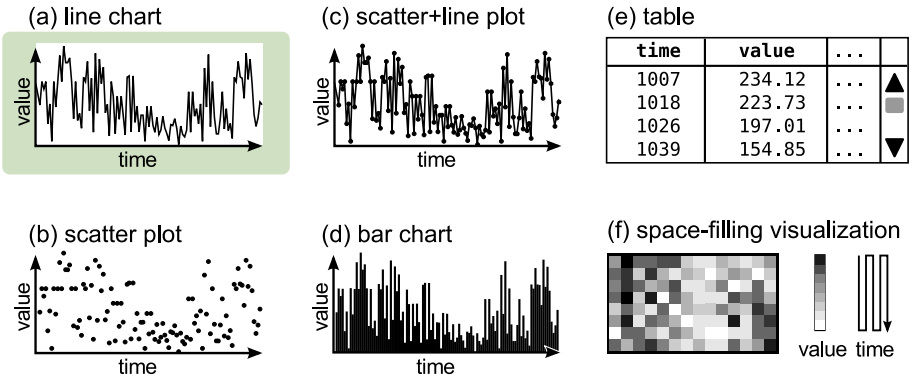


Figure 4.1.: Common data visualizations, compared to line charts.

space-filling visualization (f) that is able to render all 112 records efficiently using only 14×8 pixels. However, it may require from an untrained user some additional cognitive effort to imagine the actual shape of the signal, i.e., as it would be displayed in a more commonly used line chart.

Even though line charts are ubiquitous, a common misconception is that selecting one tuple per horizontal pixel is sufficient to visualize a larger data set [21]. Section 4 will disprove this assumption, providing an in-depth analysis of the line rendering process from a data reduction point-of-view.

4.2. Data Aggregation Model

Chapter 3 introduced the general approach to extend an incoming visualization-related query with an additional data reduction, i.e., by first grouping the data by pixel columns, and secondly computing aggregated values for each group. The following section provides a detailed description of this data aggregation model, by first discussing the boundary aggregation that is necessary for grouping the data, and secondly defining the types of data aggregation considered in this work.

4.2.1. Boundary Aggregation

A group aggregation of an original query $Q(t, v)$ depends on the two boundary values t_{start} and t_{end} of the data column t of Q, i.e., on the first and last timestamp in the entire acquired dataset. The two values are used by the horizontal geometric transformation function $f_x(t) \rightarrow \mathbb{R}_w$ (cf. Equation 2.1),

Listing 4.1: Grouping function using boundary subqueries.

```
floor($w * (t - (SELECT min(t) FROM Q))      -- project data to [0,1] to [0,w]
        / (SELECT max(t) - min(t) FROM Q)) -- and discretize to {0,1,2,...,w}
```

for a predefined width w of the visualization. The function f_x is then used by the grouping function $f_g(t) \rightarrow \mathbb{N}_w$ (cf. Equation 3.1), to compute the aggregation group keys $k = f_g(t) = \lfloor w \cdot (t - t_{start})/dt \rfloor$ with $dt = t_{end} - t_{start}$. The boundary values can be acquired and incorporated in the query rewriting process in two different ways.[1]

1. A DVS issues a preceding boundary query $G_{t_{start} \leftarrow min(t), t_{end} \leftarrow max(t)}(Q)$ to determine t_{start}, t_{end}, and thus dt, and subsequently uses the obtained values directly inside the rewritten query.

2. A DVS defines t_{start} and dt as inline subqueries of the rewritten query, i.e., $t_{start} = G_{t_{start} \leftarrow min(t)}(Q)$ and $dt = G_{dt \leftarrow max(t) - min(t)}(Q)$.

Depending on the capabilities of the used SQL database either option may provide a faster total query answer time. The SQL implementation of the subquery-based grouping function is shown in Listing 4.1, and was already demonstrated in Section 3.3.1 (cf. Listings 3.1 and 3.2).

The listings show how to compute the group keys k for the aggregation subquery, using the described grouping function f_g.

$$f_g(t) = \lfloor w \cdot (t - G_{t_{start} \leftarrow min(t)}(Q))/G_{dt \leftarrow max(t) - min(t)}(Q) \rfloor \tag{4.1}$$

For brevity, the remainder of this work denotes this grouping function in relational algebra expressions as $f_g(t)$ and in SQL listings as `$f_g(t)`. In the following, f_g is used to determine the aggregation groups for several types of data aggregations.

4.2.2. Simple Aggregation

A simple form of data reduction is to compute an aggregated value and an aggregated timestamp using the aggregation functions *min*, *max*, *avg*, *median*,

[1] In the following, as already listed in the Notations of this document (cf. page vii), this work uses the common relational algebra notations π for projection, σ for selection, and denotes grouping and data aggregation as $G_{<AggregationFunctions>}(< source >)$ or ${}_{<GroupingFunctions>}G_{<AggregationFunctions>}(< source >)$. The notation ${}_{<GroupKeys>}G_{<GroupKey \leftarrow GroupingFunction>,<AggregationFunctions>}(< source >)$ is used to explicitly include the computed group keys in the query result.

or *mode*. The resulting data reduction queries on a time series relation $Q(t, v)$, using the grouping function $f_g(t)$ and two aggregation functions f_t and f_v, can be defined in relational algebra.

$$_{f_g(t)}G_{f_t(t),f_v(v)}(Q) \tag{4.2}$$

The queries in Listings 3.1 and 3.2 already used this simple form of aggregation, selecting a minimum and thus *first timestamp* and a corresponding *average value* to model a piecewise aggregate approximation (PAA) [69]. While this effectively reduced the size of the query result to $w = 200$ tuples, it also resulted in a heavily distorted shape of the original time series (cf. Figure 3.4 on page 58).

Any averaging function, i.e., using *avg*, *median*, or *mode*, will produce similarly erroneous results, by flattening the shape of the signal through averaging out the minima and maxima. Consequently, to preserve the shape of the time series, the data aggregation must incorporate the important extrema of each group, i.e., those records with the minimum value v_{min} or maximum value v_{max} per group. Unfortunately, the grouping semantics of the relational algebra forbids a selection of non-aggregated values using the simple aggregation described above.

4.2.3. Value-Preserving Aggregation

To select from a time series relation $Q(t, v)$ the corresponding original tuples that correspond to the determined aggregated values, these values must be joined again with the underlying time series Q. Therefore, a *value-preserving aggregation* replaces one of the aggregation functions with the time-based group key (result of f_g) and joins the aggregation result with Q on that group key and on the aggregated value or timestamp. In particular, the following query

$$\pi_{t,v}(Q \bowtie {}_{f_g(t)=k \wedge v=v_g}({}_kG_{k \leftarrow f_g(t), v_g \leftarrow f_v(v)}(Q))) \tag{4.3}$$

selects the corresponding timestamps t for each aggregated value $v_g = f_v(v)$, and the following query

$$\pi_{t,v}(Q \bowtie {}_{f_g(t)=k \wedge t=t_g}({}_kG_{k \leftarrow f_g(t), t_g \leftarrow f_t(t)}(Q))) \tag{4.4}$$

selects the corresponding values v for each aggregated timestamp $t_g = f_t(t)$.

Note that these queries may select more than one tuple per group, if there are tuples with duplicate values or timestamps per group, corresponding to the aggregated values v_g or timestamps t_g. However, for most of the considered

sensor data, each record of a single time series has a unique timestamp and uses a real number with several fractional digits to represent the measured value. Therefore, this work considers the number of duplicates to be very low, i.e., less than 5%. In scenarios with more significant duplicate ratios, the described queries (4.3) and (4.4) need to be encased with duplicate-removing aggregation operators to ensure appropriate data reduction rates, as described later in Section 4.7.1.

4.2.4. Composite Aggregation

In addition to the described queries (4.2), (4.3), and (4.4) that use a single aggregated value per group, the query rewriter may also define composite queries based on several aggregated values per group. In the relational algebra, such queries can be modeled as a union of two or more aggregating subqueries that use the same grouping function. Alternatively, queries (4.3) and (4.4) can be altered to select multiple aggregated values or timestamps per group, and the join predicates can be altered, such that all different aggregated values or timestamps are correlated to either their missing timestamp or their missing value.

4.3. Stratified Sampling Aggregation

Using the described value-preserving aggregation allows the query rewriter to define simple forms of *systematic sampling*, e.g., selecting every first tuple per group using the following query.

$$\pi_{t,v}(Q \bowtie_{f_g(t)=k \wedge t=t_{min}} ({}_kG_{k \leftarrow f_g(t), t_{min} \leftarrow min(t)}(Q)))$$

Moreover, query-level *random sampling* can be conducted using a custom aggregation function $random(t)$ that selects one random timestamp from the set of timestamps in the aggregation window, i.e., using the following query.

$$\pi_{t,v}(Q \bowtie_{f_g(t)=k \wedge t=t_{rand}} ({}_kG_{k \leftarrow f_g(t), t_{rand} \leftarrow random(t)}(Q)))$$

In general, stratified random sampling in relational databases requires the input records to be sorted and numbered, e.g., by computing a numbering using the commonly available window function `row_number` $:\rightarrow \mathbb{N}$. A given row number n of each record (n, t, v) can then be compared with a random number $r \in \{min(n), ..., max(n)\}$, which can be computed for each aggregation group using the commonly available random function `random` $:\rightarrow [0, 1]$. The corresponding

Listing 4.2: Value-preserving random aggregation query.

```
WITH Q_n AS (
  SELECT t,v,                          -- original query provides
         row_number()                  -- additional row numbers
         OVER(order by t) AS n         -- for a given order
  FROM Q
)
SELECT t,v FROM Q_n JOIN (             -- the numbered records are joined
  SELECT $f_g(t) AS k, min(n) +        -- with random numbers between 1st
         floor(random()*count(*)) AS r -- and last row number
  FROM Q_n GROUP BY k                  -- per group
) AS A_rand ON  A_rand.k = $f_g(t)     -- joining on matching group keys and
           AND r = n                   -- on matching random and row numbers
```

value-preserving stratified random sampling aggregation on a numbered input relation $Q(n, t, v)$ is exemplified as SQL query in Listing 4.2 and defined as follows.

$$\pi_{t,v}(Q \bowtie_{f_g(t)=k \wedge n=r} ({}_kG_{k \leftarrow f_g(t), r \leftarrow min(n)+\lfloor random() \cdot count(*) \rfloor}(Q)))$$

In practice, any considered data aggregation or sampling can moreover be combined with system-inherent types of random sampling, e.g., using the `TABLESAMPLE` clause defined by the SQL:2003 standard [10, 47].

However, while working with small random samples of the data may provide quick approximating answers to visualization-related queries, the evaluation in Section 7.1 will show that random sampling is not appropriate for data visualizations, in particular for line charts.

4.4. MinMax Aggregation

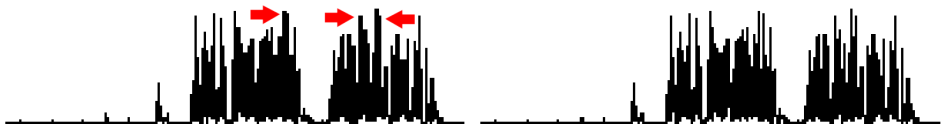
To preserve the shape of a time series, the first intuition is to groupwise select the vertical extrema, denoted as *min* and *max tuples* in this work. The corresponding *MinMax* aggregation is a composite value-preserving aggregation, defined as follows and exemplified by the SQL query in Listing 4.3.

$$\pi_{t,v}(Q \bowtie_{f_g(t)=k \wedge (v=v_{min} \vee v=v_{max})} (Q_a)$$
$$Q_a = {}_kG_{k \leftarrow f_g(t), v_{min} \leftarrow min(v), v_{max} \leftarrow max(v)}(Q)$$

The query projects existing timestamps and values from a time series relation $Q(t, v)$, matching them with the aggregated values v_{min} and v_{max}. Thereby,

Listing 4.3: Value-preserving MinMax aggregation query.

```
SELECT t,v FROM Q JOIN (
  SELECT $f_g(t) AS k,                          -- define group key
         min(v) AS v_min, max(v) AS v_max       -- compute min and max values
  FROM Q GROUP BY k                             -- for each group k
) AS Q_a ON   k = $f_g(t)                       -- join on the group key
         AND (v = v_min OR v = v_max)           -- and the computed values
```



(a) resulting image (b) expected image

Figure 4.2.: Visualization of original query result and MinMax query result.

both aggregated values are matched with the original records, using a join of the aggregation result Q_a with Q, based on the determined group keys k and by comparing the values v in Q either with v_{min} or v_{max} from Q_a.

Figure 4.2a shows the resulting visualization, which now closely matches the expected visualization of the raw data in Figure 4.2b. Section 4.5.1 will later provide an explanation for the remaining pixel errors; indicated by the arrows in Figure 4.2a.

4.5. M4 Aggregation

The composite value-preserving *MinMax* aggregation focuses on the vertical extrema of each pixel column, i.e., of each corresponding time span. There are already existing approaches that consider selecting vertical extrema for the purpose of data reduction or for data analysis [43, 38], but most of them only partially consider the implications for data visualization and neglect the final projection of the data to discrete screen pixels.

In this regard, a line chart that is based on a reduced dataset, will always *omit lines* that would have connected the not selected tuples, and it will always *add new approximating lines* to bridge the not selected tuples between two selected consecutive tuples. Thereby, if the resulting real-valued errors of the derived chart geometry are small, the subsequent discretization of the geometry from

$\mathbb{R}^2$ to the pixels in $\mathbb{N}^2$, conducted by the rendering process, may be able to mitigate or eliminate the approximation errors. This effect is the underlying principle of the effectiveness of the proposed visualization-driven data reduction techniques.

Intuitively, one may expect that selecting the *top* and *bottom* tuples $(t_{bottom}, min(v))$ and $(t_{top}, max(v))$, from exactly w groups, is sufficient to derive a correct line chart. But this intuition is elusive, and this form of data reduction – provided by the MinMax aggregation – does not guarantee an error-free line chart of a time series $Q(t, v)$. It ignores the important *first* and *last* tuples $(min(t), v_{first})$ and $(max(t), v_{last})$ of each group. The M4 aggregation additionally computes these two tuples, and Section 4.5.1 will provide a detailed discussion, how M4 surpasses the MinMax intuition. The M4 aggregation is defined as follows.

$$\begin{aligned} &\pi_{t,v}(Q \bowtie_\theta (Q_a)) \\ &Q_a = {}_kG_{k \leftarrow f_g(t), v_{min} \leftarrow min(v), v_{max} \leftarrow max(v), t_{min} \leftarrow min(t), t_{max} \leftarrow max(t)}(Q) \\ &\theta \leftarrow f_g(t) = k \wedge (v = v_{min} \vee v = v_{max} \vee t = t_{min} \vee t = t_{max}) \end{aligned} \tag{4.5}$$

M4 is a composite value-preserving aggregation (cf. Section 4.2) that groups a time series relation $Q(t, v)$ into w equidistant time spans, using the grouping function f_g (cf. Equation 3.1), such that each group exactly corresponds to one pixel column in the visualization. For each group, M4 then computes the four aggregated values $min(v)$, $max(v)$, $min(t)$, and $max(t)$ – hence the name M4 – and then joins the aggregated data with the original time series to add the missing timestamps t_{bottom} and t_{top} and the missing values v_{first} and v_{last}. Using a value-preserving aggregation is required, since the four extrema may be found in four different records of the original data, i.e., for most datasets $t_{bottom} \neq t_{top} \neq min(t) \neq max(t)$ and $v_{first} \neq v_{last} \neq min(v) \neq max(v)$.

Listing 4.4 presents an example query using the M4 aggregation. This SQL query is very similar to the MinMax query in Listing 4.3, adding only the $min(t)$ and $max(t)$ aggregations and the corresponding join predicates for the computed *first* and *last* timestamps. Figure 4.3 depicts the resulting visualization, which is now equal to the visualization of the unreduced underlying time series.

4.5.1. Aggregation-Related Pixel Errors

The inclusion of the *first* and *last* tuples is important for deriving the correct pixels of a line chart. To compare the utility of the M4 and MinMax aggre-

Listing 4.4: M4 query, extracting the first, last, bottom, top tuples.

```
SELECT t,v FROM Q JOIN (
  SELECT $f_g(t) as k,                          -- define group key
         min(v) as v_min, max(v) as v_max,      -- compute bottom and top values
         min(t) as t_min, max(t) as t_max       -- and 1st and last timestamps
  FROM Q GROUP BY k                             -- for each group key
) as Q_a ON    k = $f_g(t)                      -- join on the group key
         AND (v = v_min OR v = v_max OR         -- and the computed values
              t = t_min OR t = t_max)           -- and the computed timestamps
```

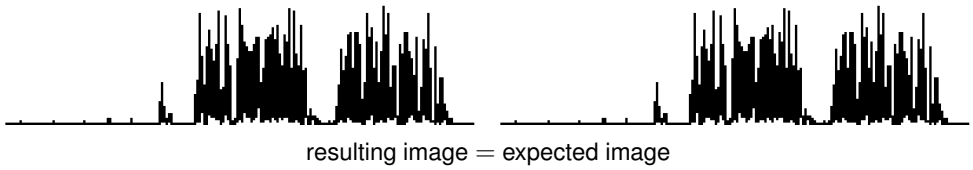


Figure 4.3.: M4 query and resulting visualization.

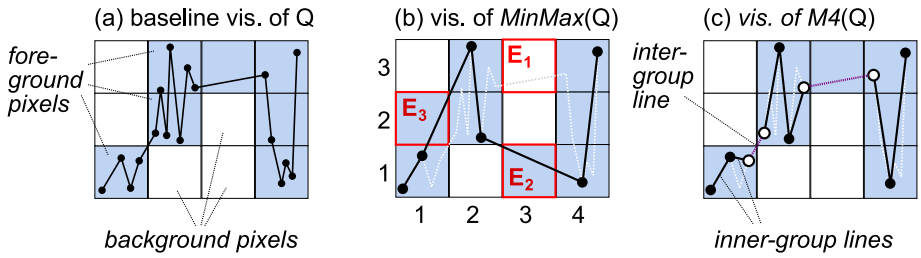


Figure 4.4.: M4 repairs MinMax-related visualization error.

gations for line charts, Figure 4.4 depicts three conceptual line charts: (a) the line chart of an unreduced time series Q, (b) the corresponding line chart of $MinMax(Q)$, and (c) the corresponding line chart of $M4(Q)$.

Note that $MinMax(Q)$ does not select the *first* and *last* tuples per pixel column and thus causes several types of line drawing errors. In Figure 4.4b, pixel $(3, 3)$ is not set correctly, since neither the start nor the end tuple of the corresponding line are included in the reduced dataset. This kind of *missing line error* E_1 is distinctly visible with time series that have a very heterogeneous time distribution, i.e., notable gaps, resulting in pixel columns not holding any data, as seen in pixel column 3 in Figure 4.4b. A missing line error is often exacerbated by an additional *false line error* E_2, since the line drawing still requires the two consecutive tuples to be connected for bridging the otherwise

empty pixel column. Furthermore, both error types may also occur in neighboring pixel columns, because the *inner-column lines* do not always represent the complete set of pixels of a column. Additional *inter-column pixels* – below or above the *inner-column pixels* – can be derived from the *inter-column lines*. This will again cause missing or additional pixels if the reduced data set does not contain the correct tuples for the correct inter-column lines. In Figure 4.4b, the MinMax aggregation causes such an error E_3 by setting the undesired pixel $(1, 2)$, derived from the *false line* between the maximum tuple in the first pixel column and the consecutive maximum tuple in the second pixel column. Note that these errors are independent of the resolution of the considered raster image, i.e., of the chosen number of groups. If the aggregated data does not include the *first* and *last* tuples for each pixel column, the renderer may not be able to draw the correct inter-column lines.

4.5.2. The M4 Upper Bound

Based on the observation of the described errors, the question arises if selecting only the four extremum tuples for each pixel column guarantees an error-free visualization. Indeed, there exists an upper bound of tuples that needs to be acquired from the raw data, to facilitate the rendering of an error-free, two-color (binary) line chart.

Definition 3. A **width-based grouping** of an arbitrary time series relation $Q(t, v)$ into w equidistant groups, denoted as $G^w(Q) = \{B_1, B_2, ..., B_w\}$, is derived from Q using the surjective grouping function $i = f_g(t) = \lfloor w \cdot (t - t_{start})/dt \rfloor$, with $dt = t_{end} - t_{start}$, for assigning any tuple of Q to the groups B_i. A tuple (t, v) is assigned to B_i if $f_g(t) = i$.

Definition 4. An **M4 aggregation** $G^w_{M4}(Q)$ selects the four extremum tuples $(t_{bottom_i}, v_{min_i})$, (t_{top_i}, v_{max_i}), (t_{min_i}, v_{first_i}), and (t_{max_i}, v_{last_i}) from each B_i of the width-based grouping $G^w(Q)$.

Definition 5. A **visualization relation** $V(x, y)$ with the data columns $x \in \mathbb{N}_w$ and $y \in \mathbb{N}_h$ contains all *foreground pixels* (x, y), representing all (black) pixels of all rasterized lines. $V(x, y)$ contains none of the remaining *background pixels* in $\mathbb{N}_w \times \mathbb{N}_h$.

Definition 6. A **visualization operator** $vis_{line}(Q) \rightarrow V$ defines, for all tuples (t, v) in Q, the corresponding *foreground pixels* (x, y) in V.

Thereby vis_{line} first uses the linear transformation functions f_x and f_y (cf. Equations 2.1) to project all tuples in Q to the coordinate system $\mathbb{R}_w \times \mathbb{R}_h$ and

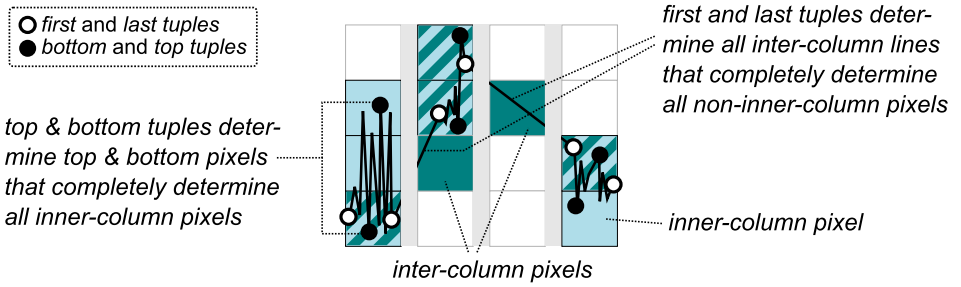


Figure 4.5.: Illustration of the M4 Theorem.

then tests for all discrete pixels and all non-discrete lines – defined by each two consecutive tuples of the projected time series data – if a pixel is *on* the line or *not on* the line. The rasterized lines adhere to the following two properties.

Lemma 2. *A rasterized line has no gaps.*

Lemma 3. *Rasterizing an inner-column line does not result in foreground pixels outside of the corresponding pixel column.*

Using the definitions above, one can now formally show that the M4 aggregation always provides the correct records for deriving the same visualization as can be derived from the raw data.

Theorem 2. *Any two-color line rendering of an arbitrary time series Q is equal to the two-color line rendering of a time series Q' that contains at least the four extrema of all groups of the width-based grouping of Q, i.e., $vis_{line}(G^w_{M4}(Q)) = vis_{line}(Q)$.*

Figure 4.5 illustrates the reasoning for the following proof of Theorem 2.

Proof. Suppose a visualization relation $V = vis_{line}(Q)$ represents the pixels of a *two-color line visualization* of an arbitrary time series Q. Suppose a tuple can only be in one of the groups B_1 to B_w that are defined by $G^w(Q)$ and that correspond to the pixel columns. Then there is only one pair of consecutive tuples $p_j \in B_i$ and $p_{j+1} \in B_{i+1}$, i.e., only one *inter-column line* between each pair of consecutive groups B_i and B_{i+1}. But then all other lines, defined by the remaining pairs of consecutive tuples in Q, must define *inner-column lines*.

As of Lemmas 2 and 3, all *inner-column pixels* can be defined from knowing the *top* and *bottom* inner-column pixels of each column. Furthermore, since f_x and f_y are linear transformations, these *top* and *bottom* pixels of each column

can be derived from the *min* and *max* tuples $(t_{bottom_i}, v_{min_i})$, (t_{top_i}, v_{max_i}) for each group B_i. The remaining *inter-column pixels* can be derived from all *inter-column lines.* Since there is only one *inter-column line* $\overline{p_j p_{j+1}}$ between each pair of consecutive groups B_i and B_{i+1}, the tuple $p_j \in B_i$ is the *last* tuple $(t_{max_i} v_{last_i})$ of B_i and $p_{j+1} \in B_{i+1}$ is the *first* tuple $(t_{min_{i+1}}, v_{first_{i+1}})$ of B_{i+1}.

Consequently, all pixels of the visualization relation V can be derived from the tuples $(t_{bottom_i}, v_{min_i})$, (t_{top_i}, v_{max_i}), $(t_{max_i} v_{last_i})$, (t_{min_i}, v_{first_i}) of each group B_i, i.e., $V = vis_{line}(G^w_{M4}(Q)) = vis_{line}(Q)$. □

The proven Theorem 2 moreover facilitates the definition of Theorem 3, regarding the cardinality of the query result.

Theorem 3. *For any time series relation Q, there exists a subset $Q' \subseteq Q$ with $|Q'| \leq 4 \cdot w$ that facilitates an error-free two-color line rendering of Q.*

No matter how big Q is, selecting the correct $4 \cdot w$ tuples from Q facilitates the creation of a pixel-perfect line chart of Q. Clearly, for the purpose of data visualization in line charts, the M4 aggregation is a data-efficient and predictable form of data reduction.

4.5.3. Sub-Pixel Grouping

A DVS may consider to use a higher data volume than $4 \cdot w$ and thus acquire additional data records with each issued query. This may reduce the overall query load. For instance, if the user requests additional details from within the initially requested time range, the visualization client may have already acquired the data and thus directly answer the user's request, without issuing another query to the database.

In this regard, there is another important property of the M4 aggregation that can be leveraged for interactive data visualizations. The visual evaluation of the M4 aggregation (cf. Section 7.1.4) will show that M4 provides error-free line charts not only from the aggregated data of $1 \cdot w$ groups, but also when using $2 \cdot w$ groups. In fact, this property also holds for any discrete factor k that is used to further group the data to $k \cdot w$ groups in the sub-pixel space of the visualization.

Theorem 4. *For any discrete group number $k \cdot w$ with $k \geq 1$, the M4 aggregation facilitates error-free two-color line renderings with a width of w pixels, i.e., $vis_{line}(G^{k \cdot w}_{M4}(Q)) = vis_{line}(Q)$ for any $k \in \mathbb{N}^+$.*

Proof. Suppose the records in any $G^k = G^{k \cdot w}_{M4}(Q)$ facilitate $V = vis_{line}(Q)$ for any $k \in \mathbb{N}^+$. Then $G^1 = G^w_{M4}(Q)$ for $k = 1$ and $G^2 = G^{2 \cdot w}_{M4}(Q)$ for $k = 2$ also facilitate V. Moreover, suppose a range $[t_0, t_w]$ is divided into w equidistant groups defined by the intersections $I^1 = \{t_0, t_1, ..., t_w\}$, including the boundary values $t_0 = t_{start}$ and $t_w = t_{end}$. Then each group has a length of $dt_w = t_1 - t_0$. Now given that G^2 uses $2 \cdot w$ likewise equidistant groups, then their length must be $dt_w/2$. Moreover, given that the intersections I^2 of G^2 likewise start at t_0 and end at t_w, then I^2 can be obtained from I^1 by adding intersections at $t + dt_w/2$ for each $t \in \{t_0, t_1, ..., t_{w-1}\}$. Therefore, the intersections I^1 are a subset of I^2. Now given that any G^k uses $k \cdot w$ equidistant groups, and since any I^k likewise starts at t_0 and ends at t_w, I^k can be obtained from I^1 by adding intersections at $t + i \cdot dt_w/k$ for each $t \in \{t_0, t_1, ..., t_{w-1}\}$ and each $i \in \{1, 2, ..., k-1\}$. Therefore, I^1 is also a subset of any I^k.

Now given that the groups for G^1 and any G^k are aligned, i.e., $I^1 \subset I^k$, and that the *min* and *max* aggregation functions are associative[2], G^1 can be derived from any G^k. But then, since G^1 facilitates V as of Theorem 2, G^k must also facilitate V and thereby provide an error-free two-color line rendering of Q at a width of w pixels. □

In practice, the above property can be leveraged to align the available zoom levels and panning steps (cf. Section 2.8.4) for an interactive visualization to the factors k of the width w of the visualization. For instance, if the DVS acquires the records for $3 \cdot w$ groups from the database, it is able to provide pixel-perfect line charts at a width of $w = 3$ pixels for the subranges listed in Table 4.1. Thereby, at a default zoom level of 100%, each 3 consecutive groups visually aggregate in one of the $w = 3$ pixel columns of the chart. When zooming in, and thus fully or partially pushing the acquired groups off the screen, fewer groups and thus less data are available for each pixel column, e.g., 2 full groups per pixel column at 150% and 1 full group per pixel column at 300%. A zoomed-in visualization can subsequently be panned inside the acquired data range. Thereby, to ensure an error-free visualization, the panning steps must be aligned with the groups, i.e., only entire group ranges may be visualized and they must start at a group offset, such that the visualized data range does not exceed the acquired data range. For the conceptual example of the $3px$-wide line chart on top of the acquired 9 records, a visualization may start at $t_{start} + k \cdot \Delta t$, with $k \in \{0, 1, 2, 3, 4, 5, 6\}$ and $\Delta t = (t_{end} - t_{start})/9$, to correctly visualize each 3 consecutive records in the 3 pixel columns, i.e.,

[2] As discussed in Section 3.4 and listed in Table 3.1.

pixel-to-group ratio (zoom level)	*safe group offset*	*resulting zoom and pan ranges*[1]
		data: $\vert (t,v)_1 \vert (t,v)_2 \vert (t,v)_3 \vert (t,v)_4 \vert (t,v)_5 \vert (t,v)_6 \vert (t,v)_7 \vert (t,v)_8 \vert (t,v)_9 \vert$
		$\vert \leftarrow t_{start}$ $t_{end} \rightarrow \vert$
$1:1$ (300%)	0	
	Δt	
	$2 \cdot \Delta t$	
	$3 \cdot \Delta t$	
	$4 \cdot \Delta t$	
	$5 \cdot \Delta t$	
	$6 \cdot \Delta t$	
$1:2$ (150%)	0	
	Δt	
	$2 \cdot \Delta t$	
	$3 \cdot \Delta t$	
$1:3$ (100%)	0	

[1] As required to obtain error-free $3px$-wide line charts for visualizing the acquired $3 \cdot w = 9$ data points.

Table 4.1.: Error-free visualizable subranges of M4 query with $3 \cdot w$ groups.

at a zoom level of 300%. Likewise, to correctly visualize 6 consecutive records in 3 pixel columns at a zoom level of 150%, the visualization must start at $t_{start} + k \cdot \Delta t$, with $k \in \{0, 1, 2, 3\}$. Finally, to visualize all 9 records in 3 pixel columns at a zoom level of 100%, the visualization must start at t_{start}.

4.6. Computational Complexity

All considered types of data aggregations (cf. Section 4.2) can be computed in linear time, i.e., in one or at most two runs over all input records, as shown by the following discussion.

Boundary Aggregation

The boundary values of a grouping-related data column may be acquired using a preceding or inline boundary aggregation. Boundary values are required to compute the aggregation group keys k, using a grouping function f_g (cf. Equation 4.1). These values are computed on the n records of the original query result Q, which in turn was computed on the base relation T. The worst case complexity to compute the boundary values, i.e., to aggregate the *min* and

max values of any data column of Q, is $\mathcal{O}(n)$, when requiring a full scan of Q. If the aggregates are transparently computable on an index of the base relation T, instead of the actually derived records of Q, the complexity is $\mathcal{O}(\log n)$.

Simple Aggregation

Given that the considered grouping facilitates processing the data aggregation as a hash aggregation, then a composite grouping aggregation, i.e., computing *min*, *max*, *count*, *avg*, or *sum* for one or more data columns of Q, has a complexity of $\mathcal{O}(n)$. The derived group key k acts as the required hash key and can be computed separately for each individual record in $\mathcal{O}(n)$, given that the boundary values of the grouping columns are already known. To avoid the second scan of the input data, k can be computed on-the-fly when computing the data aggregation.

Value-Preserving Aggregation

Composite value-preserving aggregations, such as the described MinMax and M4 aggregation, first compute a simple composite grouping aggregation and join the result again with the underlying data. The simple aggregation can be computed in $\mathcal{O}(n)$, as described above. The subsequent join is an equi-join, since all join conditions are equality predicates. This equi-join matches n tuples in Q with up to $m = 4 \cdot w$ aggregation tuples. Equi-joins can be efficiently computed using a hash-join in $\mathcal{O}(n + m)$. However, the number of groups w does not depend on n and is inherently limited by physical display resolutions, e.g., $w = 5120$ pixels for latest UHD+ displays. Therefore, the described composite value-preserving aggregation in general and the MinMax and M4 aggregations in particular have a complexity of $\mathcal{O}(n)$.

4.7. Alternative Implementations

Section 4.5 defined the M4 aggregation as a query in the relational algebra, exemplified with a supplementary SQL implementation. The described queries use a value-preserving aggregation (cf. Section 4.2) that may result in selecting more than $4 \cdot w$ records. This section will now provide an alternative, duplicate-free version of the M4 aggregation, defined in the relational algebra, exemplified as SQL query, and eventually described as a generic procedural algorithm.

4.7.1. Duplicate-Free M4

As stated before, the M4 aggregation may be selecting more than 4 records per group if the recorded values v are integer numbers of very low variety or if there are a lot of duplicate timestamps t. Thereby, for any value-preserving aggregation, the join of the original data with the aggregated values will output – for each group – all records whose timestamp or value matches with one of the aggregated values or timestamps. If within one group dozens of records have the same value v', and v' is also either a minimum or maximum value in that group, then all records with $v = v'$ are included in the query result. Theoretically, a value-preserving aggregation may even produce a query result that includes all input records. The problem can be mitigated by removing the records with the duplicated timestamps and values as follows.

1. The maxima of each group may be computed as already defined for the M4 aggregation, i.e., using the following subquery.

$$A_{M4} = {}_{k}G_{k \leftarrow f_g(t), t_{min} \leftarrow min(t), t_{max} \leftarrow max(t), v_{min} \leftarrow min(v), v_{max} \leftarrow max(v)}(Q)$$

2. The aggregated values and timestamps are correlated separately for each data column, i.e., for t and v. For each set of duplicated records, i.e., having either a shared timestamp or value, only the record with the corresponding maximum value or timestamp is included in the query result.

$$Q_t = {}_{k,t}G_{t,v \leftarrow max(v)}(Q \bowtie {}_{f_g(t)=k \wedge (t=t_{min} \vee t=t_{max})} A_{M4})$$

$$Q_v = {}_{k,v}G_{t \leftarrow max(t),v}(Q \bowtie {}_{f_g(t)=k \wedge (v=v_{min} \vee v=v_{max})} A_{M4})$$

3. The final duplicate-free query result is the union of Q_t and Q_v.

$$Q_{M4}^{nodup} = Q_v \cup Q_t$$

Note that the union in the relational algebra automatically removes additionally duplicated records included in both Q_t and Q_v. In SQL, the described duplicate removal can be implemented as shown in Listing 4.5. This duplicate removal can also be used either as a replacement or on top of the normal M4 query (cf. Listing 4.3). Since the M4 aggregation is a value-preserving aggregation including only actual records of the raw data, a subsequent duplicate removal on top of an M4 query will produce the same result as the duplicate-free version of M4, i.e., $Q_{M4}^{nodup}(Q_{M4}) \equiv Q_{M4}^{nodup}(Q)$.

Listing 4.5: M4 aggregation query with duplicate removal.

```
WITH A_m4 AS (
  SELECT $f_g(t) as k,                          -- define key
         min(v) as v_min, max(v) as v_max,      -- get min,max
         min(t) as t_min, max(t) as t_max       -- get 1st,last
  FROM Q GROUP BY k                             -- group by k
)
SELECT t,max(v) FROM Q JOIN A_m4                -- get max value
         ON k = $f_g(t)                         -- in each group
         AND (t = t_min OR t = t_max)           -- for 1st and last records
         GROUP BY k,t                           -- requires grouping by t
UNION
SELECT max(t),v FROM Q JOIN A_m4                -- get max timestamp
         ON k = $f_g(t)                         -- in each group
         AND (v = v_min OR v = v_max)           -- for 1st and last records
         GROUP BY k,v                           -- requires grouping by v
```

4.7.2. The M4 Algorithm

The M4 aggregation can also be implemented procedurally. Thereby, the correlating join of the value-preserving aggregation can be avoided by aggregating complete tuples instead single values or timestamps. Aggregation of complete tuples is not supported by the relational algebra. The resulting *M4 algorithm* is listed as Algorithm 4.2. For unsorted input it also requires using a *getBoundaries* function, listed as Algorithm 4.1. For sorted input, i.e., sorted by t in ascending order, the boundaries are determined by the first and last input record.

The Algorithm operates as follows. From a list of n input records $S = ((t_1, v_1), ..., (t_n, t_n))$, the algorithm first computes the boundary values to determine t_{start} and dt, required for the grouping function. Subsequently, it iterates over the elements of S to determine the index values i of each extremum tuple. The found index values i are stored in a map $A : k \to i$ for each computed compound group key k, with $A(k)$ defining the index i of the *first* tuple of group k and likewise $A(k+1)$ the index of the *last* tuple, $A(k+2)$ the index of the *bottom* tuple, and $A(k+3)$ the index of the *top* tuple of group k. Finally, the determined indices $i \in A$ are used to build the result set R, which eventually contains those elements $(t_i, v_i) \in S$ that are referenced by the index map A.

Note that R is a duplicate-free set and the algorithm does not compute new records, since it only operates on indices. At most $4 \cdot w$ indices are maintained

Algorithm 4.1 *getBoundaries* function for unsorted input.

function GETBOUNDARIES(S) ▷ S: list of n input records $((t_1, v_1), ..., (t_n, v_n))$
  $(t_{start}, t_{end}) \leftarrow (t_1, t_n)$ ▷ initialize boundary values
  for $i \leftarrow 1, n$ **do**
    $t_{start} \leftarrow \min(t_i, t_{start})$ ▷ update left boundary
    $t_{end} \leftarrow \max(t_i, t_{end})$ ▷ update right boundary
  end for
  return (t_{start}, t_{end})
end function

Algorithm 4.2 *M4 algorithm* for computing a duplicate-free M4 aggregation R from a list of sorted or unsorted records S.

input: S list of n records $((t_1, v_1), ..., (t_n, v_n))$
  w pixel width of the line chart
variables: A index map $k \rightarrow i$ with $k \in \mathbb{N}$ and $i \in \{1, 2, ..., n\}$
output: R set of up to $4 \cdot w$ output records

procedure M4(S, w, A, R)
  if S is sorted by t **then**
    $(t_{start}, t_{end}) \leftarrow (t_1, t_n)$
  else
    $(t_{start}, t_{end}) \leftarrow$ getBoundaries(S)
  end if
  $dt \leftarrow t_{end} - t_{start}$
  for $i \leftarrow 1, n$ **do**
    $k \leftarrow 4 \cdot \lfloor w \cdot (t_i - t_{start})/dt \rfloor$ ▷ compute group key k
    if $t_i < t_{A(k)}$ **then** $A(k) \leftarrow i$ ▷ update *first* index for k
    else if $t_i > t_{A(k+1)}$ **then** $A(k+1) \leftarrow i$ ▷ update *last* index for k
    end if
    if $v_i < v_{A(k+2)}$ **then** $A(k+2) \leftarrow i$ ▷ update *min* index for k
    else if $v_i > v_{A(k+3)}$ **then** $A(k+3) \leftarrow i$ ▷ update *max* index for k
    end if
  end for
  for all $i \in A$ **do**
    $R \leftarrow R \cup \{(t_i, v_i)\}$ ▷ add i^{th} record to the result set
  end for
end procedure

in A and thus at most $4 \cdot w$ records will be included in the result set R. Also note that, in practice, the order in which records are added to R may be determined by the order of the indices in A, which will generally not correspond to a predefined temporal order of the input records in S. Therefore, the output R is defined as generic unordered set. In practice, the algorithm may be altered to produce sorted output, i.e., respecting the order of the input records S, through sorting the computed index values in $i \in A$ by their value, before adding the records (t_i, v_i) to the result set R.

4.8. Comparison to Existing Approaches

Section 4.2 already described averaging, systematic sampling and random sampling as measures for data reduction. They provide some utility for time series dimensionality reduction in general, and may also provide useful data reduction for certain types of visualizations. However, for rasterized line visualizations, the proven Theorem 2 shows the necessity of selecting the correct *min*, *max*, *first*, and *last* tuples. As a result, any averaging and systematic or random sampling approach will fail to guarantee optimal results for line visualizations, if it does not ensure that these important tuples are included.

Compared to the present approach, the most competitive approaches are times series dimensionality reduction (TSDR) and line simplification techniques that rely on geometrical distance measures, defined between each three consecutive points of a line (cf. Section 2.7.2, Figure 2.9).

However, when visualizing a line using discrete pixels, the main shortcoming of the described measures is their generality. They are geometric measures, defined in $\mathbb{R} \times \mathbb{R}$, and do not consider the discontinuities in discrete pixel space, as defined by the cutting lines of two neighboring pixel rows or two neighboring pixel columns. For the present approach, the actual visualization determines the approximation quality of the considered data reduction. Therefore, the approximation quality is measured by comparing the visualization of the original data with the visualization of the reduced data as follows.

4.8.1. Visualization Quality

Two images of the same size can be easily compared pixel by pixel, based on their luminance data $A = (a_{1,1}, ..., a_{w,h})$ and $B = (b_{1,1}, ..., b_{w,h})$, e.g., with $a, b \in \{0, 1, ..., 255\}$ if the luminance is represented as $8bit$ integer numbers. A simple, commonly used error measure is the mean squared error $MSE =$

$\frac{1}{wh}\sum_{x=1}^{w}\sum_{y=1}^{h}(a_{x,y}-b_{x,y})^2$. However, Wang et al. have shown [102] that MSE-based measures, including the commonly applied peak signal-to-noise ratio (PSNR) [29], do not approximate the model of human perception very well. They developed the Structural Similarity Index (SSIM) for comparing image datasets, based on the following properties of the luminance datasets A and B of two images.

- The *average* values μ_A and μ_B of A and B
- The *variance* values σ_A^2 and σ_B^2 of A and B
- The *covariance* σ_{AB} of A and B
- The *dynamic range* $L = L_{max} - L_{min}$ of the luminance, e.g., $L = 255_{white} - 0_{black} = 255$

To prevent division by zero, the dynamic range L is combined with two predefined variables $k_1 = 0.01$ and $k_2 = 0.03$ in the stabilization variables $c_1 = (k_1 \cdot L)^2$ and $c_2 = (k_2 \cdot L)^2$. The SSIM is then defined as follows.

$$SSIM(A,B) = \frac{(2 \cdot \mu_A \cdot \mu_B + c_1) \cdot (2 \cdot \sigma_{AB} + c_2)}{(\mu_A^2 + \mu_B^2 + c_1) \cdot (\sigma_A^2 + \sigma_B^2 + c_2)}$$

It yields a similarity value between 1 and -1, whereby the normalized distance measure between the two luminance data sets A and B, and thus the corresponding visualizations, is defined as follows.

$$DSSIM(A,B) = \frac{1 - SSIM(A,B)}{2}$$

In Chapter 7, this work uses the $DSSIM$ function to evaluate the quality of the visualization of the reduced data set, compared to the original visualization of the underlying unreduced dataset.

4.8.2. Compliance with Data Reduction Goals

Considering the data reduction goals for this work (cf. Section 1.2), Table 4.2 lists the corresponding properties for related TSDR approaches and common line simplification algorithms, comparing them to the presented M4 and MinMax aggregation. The table lists the following properties.

1. The required input parameter, i.e., how the user can control the data reduction process.

algorithm	*complexity*	*input parameter*	*predictable*	*transparent*	*approx. quality*
random sampling	$\mathcal{O}(n)$	number of points	yes	yes	- -
PAA [67]	$\mathcal{O}(n)$	window size	yes	partially	- -
APCA [68]	$\mathcal{O}(n)$	max. error	no	partially	-
ReuWi [89]	$\mathcal{O}(n)$	max. error	no	yes	o
RDP [88, 30]	$\mathcal{O}(n^2)$	min. distance	no	yes	+
RDP' [54]	$\mathcal{O}(n \log n)$				
VisWy [100]	$\mathcal{O}(n \log n)$	number of points	yes	yes	+
MinMax	$\mathcal{O}(n)$	width	yes	yes	+
M4	$\mathcal{O}(n)$	width	yes	yes	++

Table 4.2.: Comparison of time-series dimensionality reduction techniques.

2. Whether or not the size of the query result is predictable from the considered input parameter.
3. Whether or not the query result is transparent to the visualization.
4. The complexity of the considered algorithm.
5. A rating of the achievable visualization quality.

Complexity vs. Quality. Many alternative approaches, such as PAA [67], APCA [68], and random sampling [26] can be computed in $\mathcal{O}(n)$ but suffer from a low approximation quality. In contrast, line simplification techniques provide a better approximation quality but are more expensive to compute. They either aim to minimize a distance measure ε for a selected data reduction rate ($min-\varepsilon$ problem), or they try to find the minimum number of points for a defined distance ε ($min-\#$ problem) [70]. Both problems have an $\mathcal{O}(n^2)$ worst case complexity [70] and are solved using heuristic algorithms. Thereby, the faster algorithms, such as the Reumann-Witkam algorithm [89] (ReuWi) can be computed in $\mathcal{O}(n)$. Similarly to APCA, ReuWi sequentially processes every point of the line only once but provides only a low approximation quality [91]. The more sophisticated and more common line simplification algorithms RDP [54, 30, 88] and VisWy [101, 100] provide a better approximation quality [52, 91] but have a complexity of $\mathcal{O}(n \cdot \log n)$. They either merge the points of the original line until some error criterion is met (bottom up) or split the line (top down), starting with an initial approximating line defined by the first and last points of the original line. The present work's aggregation-based

data reduction, including M4, has a complexity of $\mathcal{O}(n)$ and provides a high approximation quality.

Predictable Cardinality. Approaches driven by error measures (APCA, ReuWi, RDP) suffer from a bad predictability of the query result and thus may still result in undesirably large query result sets. To ensure a predictable cardinality of the query result, the other approaches incorporate a user-defined limit, such as the desired number of data points (random sampling, VisWy), an aggregation window size (PAA), or the width of the visualization (MinMax, M4).

Data Transparency. All listed approaches reduce the data by deriving a tuple (t', v') for one or more data subsets of the original time series relation $Q(t, v)$. Consequently, they are preserving the schema of the original query. However, averaging approaches (APCA, PAA) are not fully transparent to the visualization client in that they provide computed values instead of actual data records. While the computed averages may make sense to the user in some scenarios, they may be unexpected in others. In this regard, line simplification (ReuWi, RDP, VisWy), random sampling, and the developed visualization-driven approaches (MinMax, M4) are better. They provide actual subsets of the data.

In summary, prior approaches are either too slow ($\mathcal{O}(n \log n)$ or worse) or do not provide a high approximation quality, since they do not consider the rendering effects in line charts, i.e., the properties of rasterized lines, for the purpose of data reduction. A detailed evaluation of the different data reduction techniques on real-world data is given in Chapter 7.

Summary

This chapter developed a specific visualization-driven data aggregation for line charts. Therefore, the chapter first discussed common data aggregation models and thereof developed the M4 aggregation. This M4 aggregation is proven to provide error-free line charts of highly reduced data subsets in linear time. M4 is defined as a relational query, implementing a value-preserving data aggregation. The chapter complements this definition with a duplicate-free extension of M4 and the likewise duplicate-free M4 algorithm, which implements the value-preserving data aggregation without requiring a subsequent correlated join of aggregated values with the original data. The chapter concludes with a comparison of M4 and related data reduction techniques, showing the superiority of the present approach.

5. Visual Aggregation for Common Visualizations

A data visualization is composed of geometric primitives and embedded text strings. Thereby, common data visualizations, such as line charts, bar charts, and scatter plots, rely on simple shapes, such as lines, rectangles, circles, or crosses. When rendering a single shape on the canvas, it may overplot previously rendered shapes that were derived from the same or even other data subsets, depending on the chart type. This chapter formalizes this general observation by first defining the smallest display units of each considered chart type, deriving a corresponding grouping for the purpose of data aggregation over single and multiple data subsets. Subsequently, based on the analysis of the rendering process inside or starting at each single display unit, this chapter defines a visualization-driven data aggregation (VDDA) of numerical datasets, for the considered basic charts and also for chart matrices. The chapter concludes with a discussion of data conversion techniques in relational databases, to moreover leverage VDDA for visualizations of categorical data.

5.1. Display Units

The number of displayable records in a visualization depends on the number of *display units* of that chart type, and thus on how each chart type separates the pixel space of the visualization canvas into such display units.

Each display unit thereby either fully contains a single or stacked mark or the mark starts at this display unit and may cover neighboring display units. Each single or stacked mark and thus each display unit of a visualization displays one or several computed values, representing a potentially larger subset of records of the entire dataset. This work defines display units as follows.

Display Unit. A display unit is the smallest necessary and largest possible, discrete fraction of the pixel space that uniquely identifies one of the limited number of containers or origins of the marks in a data visualization.

The number of display units is limited by the size and structure of the visualization. A unique display unit is defined by one or several *discrete* coordinate values of one or several visualization-specific *layout dimensions*. Examples of display units are the following.

Pixel The display units of a scatter plot are the $w \cdot h$ single pixels (x, y) within the layout dimensions $x \in \mathbb{N}_w$ and $y \in \mathbb{N}_h$ of the scatter plot. The marks, i.e., the geometric shapes, in a scatter plot are freely positioned in the 2D pixel space, starting at any pixel $(x, y) \in \mathbb{N}_w \times \mathbb{N}_h$.

Table cell The cell of a table contains a padded text string to represent the data points underlying that cell. The unique coordinate (k_u, k_v) of a table cell depends on the size of the table and the size of the cells, i.e., $k_u \in \mathbb{N}_u$, and $k_v \in \mathbb{N}_v$, with $u = \lfloor w/w_{cell} \rfloor$ and $v = \lfloor h/h_{cell} \rfloor$.

Full-height bar The set of $w_{bar} \cdot h$ pixels for a full-height bar, including margins, is the display unit of a non-stacked bar chart. It can display up to three values to represent the underlying group of records, using a variable *height* of the rendered bar, the *color* of the bar, and a bar *label*. The unique coordinate k of this display unit depends on the width of the chart and the width, including margins, of the bars, i.e., $k \in \mathbb{N}_u$, with $u = \lfloor w/w_{bar} \rfloor$.

A display unit is not necessarily the same as the mark used to represent one or more data points. For instance, the marks in a scatter plot, such as rectangles or circles, are usually larger than one pixel and often require many neighboring pixels to be filled. Thereby, they overplot already drawn marks or parts of marks.

Nevertheless, for most visualizations, there is a strong correlation between marks and display units, whereby the *maximum width* and *height* of bars, cells, and marks in general, can be denoted as w_{bar}, h_{bar}, w_{cell}, h_{cell}, w_{mark}, and h_{mark}. If not stated otherwise, these measures correspond to the sizes of the underlying display units, including their margins.

Table 5.1 classifies the considered basic chart types into several chart type categories, based on the way they define display units in the 2D pixel space of a visualization. The listed categories are the following.

1D Charts are basic charts, where the visual marks are laid out horizontally *or* vertically along one layout dimension k, with $k \in \mathbb{N}_g$, and $g = \lfloor w/w_{mark} \rfloor$or

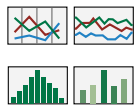
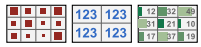
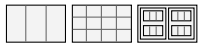
Chart Type Category	*Display Units*
1D Charts  line charts, bar charts	one or more **pixel columns** or **rows** (size depends on grid size or mark size)
 space-filling visualizations	(each pixel considered as **pixel column**)
2D Plots (overlapping shapes) 	single **pixels** (each pixel can be origin or center of a larger mark)
2D Grids (no overlap) 	**grid cells** (one cell displays one or more values to summarize the represented data)
Stacked Marks 	**non-stacking** pixel columns or rows (one for each stack, similar to 1D Charts)
Stacked Charts (chart matrices) 	**grid cells** + **nested** display units (depending on the basic chart type)
Multidimensional Grids 	**stack/grid cells** (horizontal, vertical, nested/hierarchical)

Table 5.1.: Display units of basic chart type categories.

$g = \lfloor h/h_{mark} \rfloor$. A small set of marks is drawn inside every single horizontal *or* vertical display unit; or across every two consecutive display units.

2D Plots are basic charts, where the visual marks are laid out horizontally *and* vertically along two layout dimensions x and y, with $x \in \mathbb{N}_w$ and $y \in \mathbb{N}_h$. At most one mark or fraction of a mark is visible inside every display unit.

The marks of 2D plots, e.g., a cross in a scatter plot or a filled circle in a bubble chart, are often larger than one pixel and may hide several neighboring pixels.While the size of the marks may significantly influence the overplotting and thus the visual aggregation, this work does not incorporate the size of marks when defining the corresponding display units. Every pixel is considered as a potential origin of a data point.

Note that space-filling visualizations are not 2D plots. Their marks and display units are single pixels, ordered sequentially in the 2D pixel space, along *one* layout dimension k, with $k \in \mathbb{N}_{w \cdot h}$. At most one mark is visible and fully

contained inside every display unit, and there is no overplotting to neighboring pixels. Therefore, space-filling visualizations are conceptually 1D charts.

2D Grids are basic charts, where marks are confined by the boundaries of a grid cell, so that overplotting may only occur inside each cell. The display units, i.e., the grid cells (k_u, k_v), are defined by a vertical and horizontal intersection of the visualization canvas into groups of equal size, i.e., $k_u \in \mathbb{N}_u$, and $k_v \in \mathbb{N}_v$, with $u = \lfloor w/w_{cell} \rfloor$ and $v = \lfloor h/h_{cell} \rfloor$.

A cell of a 2D grid can display more than one value, e.g., using padded text to represent one data column and a varying background color to represent another data column.

Stacked Marks are a special form of 1D charts with spatially composed marks that represent several correlated data columns. Thereby, the position of the marks of one data column depends on the position of the marks of the other data columns. A single stack of marks is displayed in each display unit of the corresponding non-stacked chart structure.

Intuitively, one could erroneously consider stacked marks as a form of 2D plots, i.e., with the coarse display units of the 1D chart further intersected into pixel rows or pixel columns. However, the different shapes of a single stack in a coarse display unit are interdependent, regarding their size and position in the stack. They share only one common pixel origin inside their coarse display unit. The pixel positions of all stacked elements are based on that origin. As a result, the coarse display units of a stacked marks chart, e.g., the full-height bars of a stacked bars chart, must not be separated any further to conduct the visual aggregation.

Stacked Charts and Chart Matrices are a combination of basic charts in a 2D grid. The charts can be combined by either stacking the complete charts (e.g., *aligned bars*) or by separating the display units of the basic charts and regrouping these units, according to their correlation in one of the shared data columns (e.g., *side-by-side bars*). The display units of a complete stacked chart are the $g \cdot u \cdot v$ display units of the $u \cdot v$ subcharts, with g being the number of display units in a single subchart. As a result, a display unit is defined by the coordinate (k, k_u, k_v), with $k_u \in \mathbb{N}_u$, $k_v \in \mathbb{N}_v$, $k \in \mathbb{N}_g$, $u = \lfloor w/w_{chart} \rfloor$, $v = \lfloor h/h_{chart} \rfloor$, and g depending on the type and size of the stacked basic charts.

Multidimensional Grids are based on multidimensional datasets where one or more data columns define a semantic grouping of the data. Using this grouping, 2D grids can be nested to define a treelike hierarchical structure, whose leaves may contain single marks or again complete basic charts.

If the grids are nested homogeneously, such that all tree leaves are on the same level and the nodes of one level use the same degree of branching, the resulting grid is visually similar to a fine-grained non-nested 2D grid, with equally sized subcharts as leaf nodes. A visual aggregation for such a structure, requires to aggregate the data in each leaf node and must be computed separately for each semantic grouping of the data, i.e., for the groups of data subsets that are displayed jointly in a leaf node. In this work, the additional categorical data columns that determine the semantic grouping are not considered for the visual aggregation. Therefore, a multidimensional grid is considered as a set of independent subcharts of one of the previously described chart types. Multidimensional grids that allow visual aggregation at the higher nesting levels, as well as inhomogeneously nested grids, and thus true treelike visualizations like treemaps [92] are out of scope of this work.

5.2. Rendering of Ordered and Unordered Data

For some visualizations, the eventually visible data is affected by the order in which data is rendered. For instance, consider Figure 5.1, which illustrates how several marks in scatter plots hide one or more underlying marks. Thereby, Figure 5.1a depicts a scatter plot that uses variable sized circles, i.e., a bubble chart, where the largest circle is rendered as one of the last marks, effectively and undesirably overplotting most of the underlying marks. Figure 5.1b shows an improved version of this plot, where the data is ordered by size of the marks, from the largest to the smallest, before being rendered. This results in all marks being at least partially visible in this specific bubble chart and demonstrates the importance of the rendering order.

The order of the data affects the rendering result not only at this macro level, but also at the pixel level, e.g., in scatter plots with single pixel marks. For instance, Figures 5.1c and 5.1d depict two different versions of a scatter plot of two datasets, where each set is assigned a specific color. When projecting data points to pixels, each pixel is effectively assigned the associated color of the last rendered data point. Previously assigned colors are completely overwritten. As a result, the rendering order determines the color of each single pixel. In Figure 5.1c, the data is ordered horizontally (by x) before rendering, while in Figure

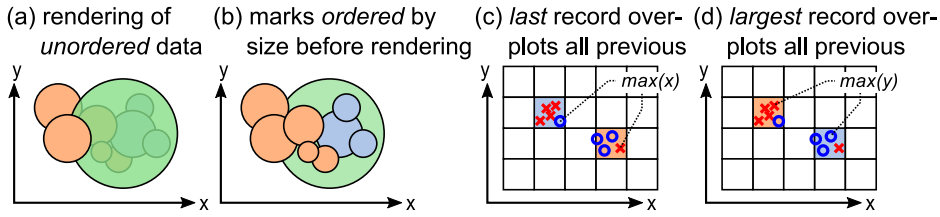


Figure 5.1.: Rendering of ordered and unordered data.

5.1d, the data is ordered vertically (by y). The different rendering orders result in the two drawn pixels having different colors in Figure 5.1d, compared to Figure 5.1c.

The above discussion of Figures 5.1a-d shows how the final visibility of large but also of small marks depends on the rendering order. Consequently, the rendering order is another important property of the considered visualization, resulting in the following implications for the visual aggregation of ordered and unordered datasets.

Unordered Data is not explicitly ordered by the visualization-related original query. The effective order of the query result, and thus the visibility of marks in the final visualization, cannot be determined beforehand. Nevertheless, a corresponding visual aggregation must anticipate a specific rendering order to approximate the rendering process.

Ordered Data is either ordered by the visualization-related original query or subsequently by the visualization framework. The order can be determined by analyzing the original query or given as a parameter to the query rewriter by the visualization client. A corresponding visual aggregation may incorporate this order to better approximate the rendering process.

As a heuristic for approximating the rendering of unordered data, a visual aggregation may consider the size of the marks, and eventually select the largest, i.e., dominating, mark per display unit. Large marks are more likely to partially or fully overplot underlying marks than smaller marks. Another order-related heuristic is to consider the values of an unordered data column as ordered, e.g., horizontally, and select the last record per display unit, according to that order.

Note that, for scatter plots with large geometric shapes, using these heuristics and even a given order do not suffice to model the rendering process as

data aggregation in the defined display units. Several overlaying shapes with the same pixel center may be visible in the final rendering. A corresponding visual aggregation would have to incorporate the cross-display-unit features of these shapes, to effectively include all visible data points in the aggregation result. However, the developed visual aggregation techniques do not leverage such cross-display-unit rendering effects. Chapter 6 will provide a solution to mitigate the problems caused by these effects. But the consideration of the entirety of rendering effects, including all kinds of overplotting of marks over marks in neighboring pixels, is out of scope of this work.

In general, an RDBMS treats data as unordered sets rather than ordered lists, and thus the records of a query result are generally considered as unordered. However, since there is usually a natural order inherent in the data, i.e., the temporal order of a time series, a DVS will often sort the acquired data before rendering. Consequently, for the purpose of data visualization and for defining a corresponding visual aggregation, the records of a time series $Q(t, v)$, as considered in this work, can be regarded as ordered by their timestamps t, independent of how the database treats the stored data.

5.3. The VDDA Query Model

Similar to the MinMax and M4 aggregation for line charts, one can define a specific form of data aggregation for each of the considered chart types, based on a general aggregation model, applicable for all types of visualizations. Therefore, the following sections describe in detail how to compose the data reduction query Q_r, to be used by the query rewriter (cf. Section 3.3.1), for each of the defined chart types (cf. Section 2.8.2).

5.3.1. VDDA Query Template

Before discussing in detail the visual aggregation of bar charts in Section 5.3.4 and of the remaining chart types in Section 5.3.5, the following section first defines a general model for the considered data aggregation queries, decomposing them into several subqueries.

Therefore, Listing 5.1 shows a complete VDDA query Q_r that is composed of several subqueries, implementing the original query, the grouping, the aggregation, and the final correlation of the aggregated values with the original data. Exactly like the M4 and MinMax aggregation, all developed VDDA queries define a value-preserving aggregation, as defined in Section 4.2.

Listing 5.1: VDDA query template.

```
WITH
Q   AS (SELECT a,b FROM T WHERE a >= $a1 AND a =< $a2), -- original query
Q_b AS (SELECT min(a) a1, max(a) - min(a) da FROM Q), -- compute boundaries
Q_g AS (SELECT *, floor( $n * (a - a1) / da) k     -- provide group keys
        FROM Q CROSS JOIN Q_b),                 -- for all n display units

A_bar AS (SELECT k,max(b) b_max,          -- compute aggregated values
          FROM Q_g GROUP BY k),           -- for each group

Q_bar AS (SELECT * FROM Q_g JOIN A_bar    -- correlate aggregated + raw data
          ON  A_bar.k = Q_g.k             -- via equi-join on group key
          AND A_bar.b_max = Q_g.b)        -- and the aggregated values

SELECT a,b FROM Q_bar                     -- remove auxiliary columns
```

Note that the query Q_r is equal to the data reduction subquery as defined for the query rewriting process in Section 3.3.1, i.e., it only models the visualization-specific data reduction and does not include the additional cardinality check and the conditional execution statements.

Listing 5.1 moreover acts as a template, allowing the DVS to model a data aggregation query for each chart type. Thereby, the visualization-specific subqueries can be replaced with appropriate subqueries for each defined basic or composite visualization. The provided query template contains the following subqueries.

Q — A subquery containing the original query, as defined by the visualization client, incorporating all parameters that are not explicitly exposed to the query rewriter.

Q_b — A subquery to compute the boundaries of the requested data range. This subquery computes the boundaries for all data columns that have to be projected to the corresponding layout dimensions of the visualization.

Q_g — A subquery that adds an additional column k to Q, containing the group key that represents the corresponding display unit for each record.

$A_{<chart>}$ A chart-specific subquery that simulates or approximates the visual data reduction of the rendering process. This subquery has as output the aggregated values and the corresponding group key. Note that the result of $A_{<chart>}$ is not yet compatible with the visualization data model, i.e., with the original query Q.

$Q_{<chart>}$ A chart-specific subquery that correlates the aggregated values in $A_{<chart>}$ with the original data in Q_g. This query produces a relation that is compatible with the corresponding visualization data model, and thus with Q.

The final result must have the same data columns as the original query, i.e., include all data columns to be displayed by the visualization. Therefore the final subquery projects only these data columns from $Q_{<chart>}$, effectively removing all auxiliary columns.

5.3.2. Visual Multidimensional Grouping

To compute group keys for basic chart types, such as bar charts, line charts, and scatter plots, only one or two numerical data columns must be projected to the layout dimensions of the visualization. Therefore, the group key subquery Q_g either uses a one-dimensional grouping function $f_g : \mathbb{R} \to \mathbb{N}_w$, or a two-dimensional grouping function $f_g : \mathbb{R} \times \mathbb{R} \to \mathbb{N}_{w \cdot h}$ as introduced in Section 3.3.1 (cf. Equations 3.1 and 3.2). Each record in Q_g then contains the original data record and one of the group keys k_1 to k_n, with n being the total number of display units, e.g., the w pixel columns of a line chart, or the $w \cdot h$ pixels of a scatter plot.

For more complex visualizations, this grouping function must incorporate the additional layout dimension of the specific visualization. Moreover, to keep the aggregation model simple, this grouping function must again compute only one key for every record, independent of the number of considered layout dimensions of the visualization. This general grouping function $f_g : \mathbb{R} \times ... \times \mathbb{R} \to \mathbb{N}$ is defined as follows.

Let $f_i(a_i)$ be a grouping function for distributing the values of a single data column a_i, to the n_i display units of a corresponding layout dimension.

$$\begin{aligned} f_i(a_i) &= \lfloor n_i \cdot (a_i - a_{i_{min}})/(a_{i_{max}} - a_{i_{min}}) \rfloor \\ &= \lfloor n_i \cdot (a_i - a_{i_{min}})/\Delta a_i \rfloor \end{aligned}$$

Then, for any number N of considered layout dimensions, one or more auxiliary functions f_i can be used to define a single multidimensional grouping function

as follows.

$$2D: \; f_g(a_1, a_2) = n_1 \cdot f_2(a_2) + f_1(a_1)$$

$$\begin{aligned} 3D: \; f_g(a_1, a_2, a_3) &= n_1 \cdot n_2 \cdot f_3(a_3) \\ &+ n_1 \cdot f_2(a_2) \\ &+ f_1(a_1) \end{aligned}$$

$$ND: \; f_g(a_1, ..., a_N) = \sum_{i=1}^{N} (f_i(a_i) \cdot \prod_{j=0}^{i-1} n_j) \tag{5.1}$$

The factors n_1 to n_N are the numbers of display units of each layout dimension. The factor $n_0 = 1$ is required for formal correctness and does not influence the result.

Example 2. *A scatter matrix* splits the visualization into several smaller subcharts, groups data subsets by the matrix columns and rows, and eventually projects each record to a pixel in a subchart. Given that all subcharts have the same pixel width w and pixel height h, and the matrix has u columns and v rows, a combined group key can be computed using the following multidimensional grouping function.

$$\begin{aligned} f_g \quad & (a, b, c, d) = \\ & w \cdot h \cdot u \cdot \lfloor v \cdot (d - d_{min})/\Delta d \rfloor \\ + \quad & w \cdot h \cdot \lfloor u \cdot (c - c_{min})/\Delta c \rfloor \\ + \quad & w \cdot \lfloor h \cdot (b - b_{min})/\Delta b \rfloor \\ + \quad & \lfloor w \cdot (a - a_{min})/\Delta a \rfloor \end{aligned}$$

In this example, the data columns a and b determine the x and y coordinates of the marks in each subchart. The data columns c and d determine which subchart each record corresponds to. The exemplified multidimensional grouping can be expressed as relational operator and implemented using SQL. Listing 5.2 shows the corresponding boundary and group key subqueries Q_b and Q_g.

In the following, a boundary subquery Q_b and a group key subquery Q_g, using a specific N-ary grouping function $f_g(a_1, a_2, ..., a_N)$, are defined for each chart type category.

1D Charts require to divide one data column a of the selected data range into n_a equidistant intervals corresponding to either the horizontal or vertical display units of the chart. Therefore, 1D charts use the following subqueries.

$$\begin{aligned} Q_b &= G_{max(a),min(a)}(Q) \\ Q_g &= \pi_{k \leftarrow f_g(a),a,b}(Q) \end{aligned}$$

Listing 5.2: 2D boundary and group key subqueries for a scatter matrix.

```
-- boundary subquery Q_b --
(SELECT min(a) as a1, max(a) - min(a) as da,
        min(b) as b1, max(b) - min(b) as db,
        min(c) as c1, max(c) - min(c) as dc,
        min(d) as d1, max(d) - min(d) as dd FROM Q)

-- group key subquery Q_g --
(SELECT Q.*, $w * $h * $u floor( $v * (d - d1) / dd )
             +    $w * $h floor( $u * (c - c1) / dc )
             +         $w floor( $h * (b - b1) / db )
             +            floor( $w * (a - a1) / da ) as k
 FROM Q CROSS JOIN Q_b)
```

2D Plots require to divide two data columns a and b of the selected data range into n_a horizontal intervals and n_b vertical intervals, resulting in $n_a \cdot n_b$ data subranges corresponding to the display units of the plot. The boundary subquery must compute the boundary values for the horizontal and the vertical value ranges. The values n_a and n_b usually correspond to the pixel width w and the pixel height h of the canvas of the plot. 2D plots use the following subqueries.

$$Q_b = G_{max(a),min(a),max(b),min(b)}(Q)$$
$$Q_g = \pi_{k \leftarrow f_g(a,b),a,b,c}(Q)$$

Space-Filling Visualizations use the same group key subquery Q_g as 1D Charts, with the number of display units defined as $n_a = w \cdot h$.

2D Grids use the same grouping and boundary subqueries as 2D Plots, with n_a defined by the number of columns and n_b defined by the number of rows of the grid, i.e., $n_a = \lfloor w/w_{cell} \rfloor$ and $n_b = \lfloor h/h_{cell} \rfloor$.

Stacked Charts require an outer grouping, similar to the grouping of 2D Grids, and an inner grouping, defined by the type of the subcharts. Thereby, stacked *1D Charts* use an additional data column c that is related to the n_c display units of the stacked subcharts, resulting in the following boundary and group key subqueries.

$$Q_b = G_{max(a),min(a),max(b),min(b),max(c),min(c)}(Q)$$
$$Q_g = \pi_{k \leftarrow f_g(a,b,c),a,b,c,d}(Q)$$

The boundary and group key subqueries for stacked *2D Plots*, e.g., scatter matrices, require yet another data column d and are defined as follows.

$$Q_b = G_{max(a),min(a),max(b),min(b),max(c),min(c),max(d),min(d)}(Q)$$
$$Q_g = \pi_{k \leftarrow f_g(a,b,c,d),a,b,c,d,e}(Q)$$

Stacked Marks require to align and combine the values ranges of a set of correlated data columns. Therefore, different stackable data columns are combined by addition, e.g., using the following subquery that combines three stackable data columns b, c, and d.

$$Q_s = \pi_{a,b,c,d,s \leftarrow (b+c+d)}(Q)$$

Alternatively, stackable records in one single data column b can be combined by computing a correlated groupwise summation of b, with the data grouped by another non-stacked data column a, e.g., using the following subquery.

$$Q_s = \pi_{a,b,c,s}(Q \bowtie ({}_aG_{a,s \leftarrow sum(b)}(Q)))$$

The latter is required if the recorded values of several data subsets are stored in a single data column b. For time series data, each data subset must moreover be identified by a key, provided by another data column c. Thereby, for each value of data column a, at most one value must exist in data column b per key in c, to ensure that the *sum* of each stack in the computed column s correctly simulates the stacking, i.e., every time series must contribute at most one record to each stack to ensure a valid visualization of the data. Given a computed stacking Q_s, the resulting boundary and group key subqueries for stacked marks are similar to those of 1D Charts.

$$Q_b = G_{max(a),min(a)}(Q_s)$$
$$Q_g = \pi_{k \leftarrow f_g(a),a,b,c,\ldots,s}(Q_s)$$

However, they use Q_s as a substitute for the original query Q, and the group key subquery Q_g moreover projects all data columns from Q_s, including the additional stacking column s.

5.3.3. Non-Visual Multidimensional Grouping

The group key subqueries Q_g for each chart type are defined for rendering one or more series jointly on one canvas of a basic chart or in several subcharts of a chart matrix. For chart matrices, the VDDA grouping function (cf. Equation

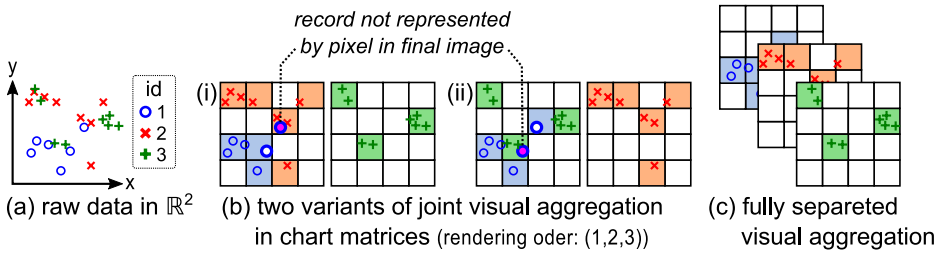


Figure 5.2.: Overplotting variants in scatter matrix.

5.1) thereby represents only one way of distributing multiple series to different subcharts of the matrix. It groups the data *linearly* according to a continuous numerical data column. However, which series are rendered in which subchart may be determined differently by the final visualization, e.g., based on the values of additional categorical data columns. Depending on which series are eventually rendered together in a single subchart, the overplotting and thus the visual aggregation result may vary, effectively showing different views on the data.

Example 3. Consider a dataset $Q(id, a, b)$ containing three data series distinguished by their different $id \in \{1, 2, 3\}$, as depicted in Figure 5.2a. The series can be distributed to a conceptual 2×1 scatter matrix in various ways, as illustrated in Figure 5.2b. The figure shows a scatter matrix (i) where series 1 and 2 are rendered jointly in one subchart, and series 3 is rendered separately. It also shows a scatter matrix (ii) where series 1 and 3 are rendered jointly, and series 2 is rendered separately. Depending on the rendering order and the distribution scheme of the series, i.e., $((1, 2), (3))$ vs. $((1, 3), (2))$, the resulting images are different. For instance, the last record of series 1 in matrix (i) is overplotted and not represented by a pixel in the corresponding image. In matrix (ii), this last record of series 1 is visible, but the second last record is overplotted.

As a result, if the rendering order and distribution scheme of different series in a chart matrix is not known or multiple series should only be overplotted dynamically, i.e., by interactively moving the focus from one series to another and thereby bringing the active series to the front and shifting other series to the back, then each series must be rendered separately and a corresponding query-level visual aggregation must also be conducted separately for each series (cf. Figure 5.2c).

In this case, the layout dimensions of the chart matrix must not be incorporated in the grouping function, and instead an additional grouping of the data by the series identifiers, e.g., by *id*, must be used. Section 3.5 already used this approach exemplarily for a multi-series line chart (cf. Listing 3.2), computing a PAA data reduction for each of the individual sensor signals.

Therefore, in the following, the visual multidimensional grouping function f_g (cf. Equation 5.1) is further modified to support semantic grouping by series identifiers in addition to visual grouping by layout dimensions. Again, the resulting function $f_g : \mathbb{R} \times ... \times \mathbb{R} \to \mathbb{N}$ must compute only one group key k for each input record to ensure the generality of the overall approach.

Let $f_g^0(a_1, ..., a_G)$ be the purely visual grouping function that groups only the considered layout dimensions, e.g., excluding the row and column dimensions of a chart matrix. Then the following grouping function f_g can be used to additionally group the data semantically by one or more key columns b_1 to b_I.

$$f_g(a_1, ..., a_G, b_1, ..., b_I) = key(b_1, ..., b_I) + m \cdot f_g^0(a_1, ..., a_G) \tag{5.2}$$

The variable $m = G_{count(*)}(\pi_{b_1,...,b_I}(Q)) \in \mathbb{N}$ is the total number of unique series, i.e., combinations $(b_1, ..., b_I)$, and the n-ary function $key \to \mathbb{N}_m$ returns a running number from 1 to m for each combination. The result of f_g is again a unique group key k that identifies the groups of records to be aggregated visually.

As demonstrated in Listing 3.2, the implementation of this semantic rather than visual grouping of the data can also be defined outside of the grouping function. In practice, the visual aggregation subquery $A_{<chart>}$ can be changed to additionally group the data by the key columns b_1 to b_I. The subsequent correlation subquery $Q_{<chart>}$ must then include the corresponding conjunctive join predicates for b_1 to b_I, i.e., $A.b_1 = Q.b_1 \wedge ... \wedge A.b_I = Q.b_I$.

To avoid defining overly complex relational algebra expressions, the remainder of this work considers any visual and semantic grouping of the data to be defined solely by the group key subquery Q_g, i.e., using a single function $f_g \to \mathbb{N}$ as defined in Equation 5.2.

5.3.4. Defining the Visual Aggregation

The principle of visual aggregation was already demonstrated by the M4 aggregation in Chapter 4. To define the visual aggregations of the other chart types, the rendering process of the considered visualization must be decomposed and analyzed. The analysis process is demonstrated by the following exemplary discussion of the rendering behavior of a bar chart.

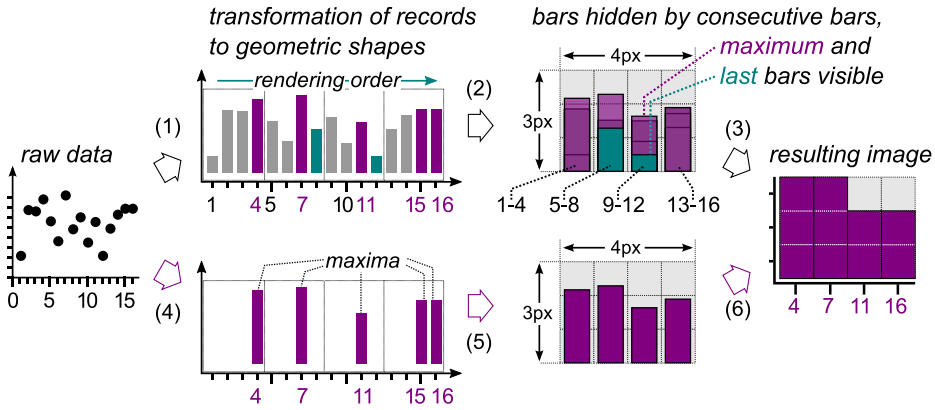


Figure 5.3.: Visual aggregation in bar charts.

Example 4. A bar chart of a single series $Q(a, b)$ displays only two data columns, i.e., the data column a, related to the horizontal layout dimension of the bar chart, and a data column b that determines the height of the bars. When visualizing the raw data Q on the visualization canvas, a bar is drawn for each record $(a, b) \in Q$, with the discrete pixel position of the bar determined by a and the pixel height of the bar determined b.

Figure 5.3 depicts this rendering process and illustrates how a resulting large number of bars visually aggregates in the pixel space of a bar chart, i.e., in $w = 4$ pixel columns. The rendering-inherent visual aggregation works as follows.

1. The raw data is transformed to geometric primitives, i.e., one bar for each data record.

2. The bars are projected to the pixel space, whereby the following overplotting behavior can be observed.

   - The bars of consecutive records in one pixel column have the same horizontal and vertical origin, e.g., in the bottom-left pixel of each column.
   - Consecutive bars overplot each other, e.g., determined by the given horizontal order of the series, whereby larger bars fully overplot smaller bars in the same pixel column.
   - After all bars of a pixel column are rendered, the eventually visible stack of overlaying bars displays the last rendered bar completely

and may also display several underlying bars partially. The stack is vertically as high as the highest bar in the corresponding pixel column.

3. Consequently, given that all bars are rendered in the same color, the final image can be considered to show only the highest bar in each pixel column.

4. From the observations above, it can be deduced that acquiring only the records corresponding to the *highest bars*,

5. and rendering only those highest bars,

6. results in the same image as obtained from rendering all bars.

A corresponding data aggregation for the visual aggregation in a bar chart is to the select the maximum record for each pixel column, i.e., using the value-preserving aggregation

$$Q_{bar} = \pi_{a,b}(Q_g \bowtie_{k_G=k \wedge b=b_{max}} ({}_kG_{k_G \leftarrow k, b_{max} \leftarrow max(b)}(Q_g))$$

on the group key subquery for 1D charts, i.e., on $Q_g = \pi_{k \leftarrow f_g(a),a,b}(Q)$. The query Q_{bar} can be implemented in SQL, as exemplarily shown in the query template in Listing 5.1.

The above example demonstrates the analysis of a basic chart type and the definition of a corresponding data aggregation and implementation in SQL. Similarly, the following section will discuss the overplotting behavior of the remaining chart types.

5.3.5. Visual Aggregation in Common Chart Types

The aggregation functions for the considered chart types are similar to the Min-Max or M4 aggregation, selecting one or more extrema for each display unit. These extrema often represent the biggest geometric shape rendered inside or starting at this display unit. In most visualizations, this biggest shape completely hides or visually dominates smaller shapes in the same display unit. In these cases, the aggregation subquery $A_{<chart>}$ computes the maximum values for each group and for each data column that is relevant for the visual aggregation, i.e., for data columns that influence the size and thus the overplotting. The correlation subquery $Q_{<chart>}$ subsequently selects the corresponding values of the other visualized data columns.

Listing 5.3: Correlated maximum aggregation $Q_{max}(t, v; id, t, v)$.

```
-- aggregation --
WITH A_max AS (SELECT k, max(t) t_max, max(v) v_max -- compute last timestamp
               FROM Q_g GROUP BY k)                 -- and highest value
-- correlation --
SELECT id,t,v FROM Q_g JOIN A_max            -- and extract the corresponding
ON  Q_g.k = A_max.k                          -- records from the raw data
AND (t_max = t OR v_max = v)
```

For brevity, in the following, the combination of a value-preserving maximum aggregation and a corresponding correlation subquery is called a *correlated maximum aggregation* (CMA) and denoted as Q_{max}.

Correlated Maximum Aggregation. A correlated maximum aggregation is defined for the data columns $\{a_1, ..., a_N\}$ of an original query Q with a subset $\{g_1, ...g_V\} \subset \{a_1, ..., a_N\}$ of visually aggregating columns as

$$Q_{max}(g_1, ..., g_V; a_1, ..., a_N) = \pi_{a_1,...,a_N}(Q_g \bowtie_\theta (A_{max}))$$

with the following aggregation subquery A_{max} and equi-join condition θ.

$$\begin{aligned} A_{max}(g_1, ..., g_V) = \quad & {}_kG_{k_G \leftarrow k, g1_{max} \leftarrow max(g_1), ..., g_{V_{max}} \leftarrow max(g_V)}(Q_g) \\ \theta \leftarrow \quad & k = k_G \wedge (g_1 = g1_{max} \vee ... \vee g_V = g_{V_{max}}) \end{aligned}$$

As a specific value-preserving aggregation, Q_{max} joins the result of an aggregation subquery A_{max} with a group key subquery Q_g on a group key k and on any of the aggregated data columns g. Q_{max} subsequently projects all original data columns from the join result. Thereby, Q_{max} and thus A_{max} are defined independently of whether or not any of the columns a_1 to a_N of Q are used by the group key subquery Q_g to compute the group keys k.

Example 5. An exemplary SQL implementation of a specific CMA $Q_{max}(t, v; id, t, v)$ for the visual aggregation of a multitude of time series $Q(id, t, v)$ is shown Listing 5.3. This particular query and thus a corresponding visualization consider the last rendered shape in each display unit, i.e., at each $t = t_{max}$, as well as each largest shape, i.e., with $v = v_{max}$, as the most important shapes in the visualization.

Except for line charts, a CMA can be defined for all of the considered chart types, to approximate or simulate the overplotting behavior of the rendering process. The CMAs Q_{max} for the considered chart types (cf. Table on page 42) are defined as follows.

Bar Charts can display two data columns a and b, with a determining the position of the bars and b determining the height of the bars. Consecutive bars overplot each other inside each display unit, with the final rendering effectively displaying only the maximum bars, i.e., the bar with the highest value of b for each display unit. The corresponding CMA is defined as $Q_{bar} = Q_{max}(b; a, b)$.

Scatter Plots use geometric shapes or single pixels as marks positioned freely in the pixel space. For each display unit, i.e., pixel, a first visual aggregation variant selects the record that produces the biggest shape, using the CMA $Q_{max}(c; a, b, c)$, with a and b projecting to the coordinates x and y of a shape, and c determining the size of the shape.

A second, alternative visual aggregation is required for scatter plots that do not map any data column to the size of the marks, i.e., using equally sized marks or single pixels. In this case, the CMA $Q_{max}(a; a, b)$ aggregates over the horizontal data column a, computing the record with the maximum a for each display unit., i.e., the *last* horizontal record per pixel. This simulates how subsequently rendered shapes overplot the previously rendered shapes starting at the same pixel.

Aligned Bars are a visual combination of several bar charts in a 2D grid. Each coarse display unit, i.e., each basic bar chart, displays the values of a single data column. A bar can only overplot bars in the same subchart. As a result, a visual aggregation can use separately aggregated values – one for each data column c_1 to c_m – to simulate the overplotting. The resulting CMA is $Q_{multibar} = Q_{max}(c_1, ..., c_m; a, b, c_1, ..., c_m)$, with a and b projecting to k_u and k_v of the 2D grid of the chart matrix.

Alternatively, if the correlated datasets are not stored as separate data columns c_1 to c_m, but as a single column c and with an additional series identifier column d, the visual aggregation can be computed using the CMA $Q_{multibar} = Q_{max}(c; a, b, c, d)$, given that the grouping function of the underlying group key subquery Q_g incorporates d to produce separate groups for each correlated dataset.

Side-by-Side Bars are a visual combination of several bar charts in a 2D grid. Each coarse display unit, i.e., a group of bars, displays at most one value per correlated dataset, i.e., per data column. Each correlated bar of a group of bars is overplotted separately. As a result, a visual aggregation can use separately aggregated values – one for each data column – to simulate the overplotting.

The resulting CMA is the same $Q_{multibar}$ as defined above for aligned bar charts.

Stacked Bars are bar charts with a set of correlated marks, i.e., bar segments, forming one composite geometric shape per display unit. For the visual aggregation, each stack of bar segments is treated as a single bar, as in a normal bar chart. The highest stack is again considered as the dominant mark, and a corresponding maximum aggregation has to select exactly those records that contribute to the highest stack in each display unit.

The height of a stack can be computed as the sum of the heights of the bar segments. Consequently, for stacked bars charts, Q_{max} requires Q_g to contain a summation column s that simulates the geometric stacking (cf. Section 5.3.2). This summation column is then used by Q_{max} to compute the visual aggregation. As an auxiliary column, it is subsequently excluded from the final query result. Therefore, the CMA for stacked bar charts is $Q_{max}(s; a, b_1, ..., b_n)$, with the summation column s modeling the stacking, the non-stacking data column a projecting to the horizontal layout dimension of the basic bar chart, and the stacked data columns b_1 to b_n determining the height of the bar segments.

If the stacked values of different data subsets are stored in a single data column b, and again distinguished using a series identifier column c, then the visual aggregation can be computed using $Q_{max}(s; a, b, c)$. Note that c must not be used for grouping the data in Q_g, since a stacked bar chart combines all data columns to a single composite mark in each display unit.

Measure Bars are basic bar charts that can visualize one additional numerical data column, using the color of the bar. Measure bars, use the CMA $Q_{max}(b; a, b, c)$, with a projecting to the horizontal layout dimension of the basic bar chart, b defining the height of the bars, and c defining the color of the bars.

Text Tables define a 2D grid. Every grid cell can display one value. When rendering several records inside the same cell, the text strings overlay each other, making the visual result unreadable to the user. For overlaid texts, there is no geometric mark that dominates the other marks. Consequently, a value must be displayed that represents the data range of the cell *semantically*, rather than *visually*. Commonly used summarizing representatives are the min, max, $first$, $last$, and mean (avg) values. From these options, this work considers the maximum value as a quasi-visual aggregation, corresponding to

the largest graphical shape that could be rendered in a table cell, instead of the textual value. To visually aggregate the data, text tables use the CMA $Q_{max}(c; a, b, c)$, with the data columns a and b projecting to the vertical and horizontal layout dimensions k_u and k_v of the 2D grid of the table, and with the aggregated maximum values of data column c displayed in the table cells.

Heat Maps define a 2D grid. Each cell of the grid is used to display one value. For heat maps that use variable-sized rectangles inside each cell, the visibility of the resulting shapes depends on the rendering order. If the heat map uses colors instead of shapes to indicate a value, only the colored table cell itself is visible. In both cases, this work again considers the record with the maximum value to represent the data. This largest or "hottest" record for each grid cell is determined using the CMA $Q_{max}(c; a, b, c)$, with a and b projecting to k_u and k_v of the 2D grid of the heat map, and with the aggregated values of the data column c determining the color of a cell or the size of the shape to be rendered inside the cell.

Highlight Tables define a 2D grid, rendering a geometric shape and a text value in each grid cell. They are a similar to text tables and heat maps and use the same CMA $Q_{max}(c; a, b, c)$, as used for heat maps to compute a maximum record per cell.

Circle Charts are simplified scatter plots. Instead of using pixels as display units, one of the layout dimensions k_u or k_v is defined by a coarse intersection of the canvas, i.e., $k_u \in \{1, 2, ..., u\}$ or $k_v \in \{1, 2, ..., v\}$, with $u = \lfloor w/w_{cell} \rfloor$ or $v = \lfloor h/h_{cell} \rfloor$. The size and the shape of the marks are fixed, i.e., circle charts can display two data columns less than scatter plots. Shapes with the same center pixel are fully overplotting each other. As a result, only one record per grid column or grid row and per pixel row or pixel column is required to draw a correct visualization. The last rendered record is again determined by the rendering order. Given that the data is ordered by a data column a, circle charts use the CMA $Q_{max}(a; a, b, c)$, with a and b determining the coordinates (x, k_v) or (k_u, y) of the circle and c defining the color of the circle.

Scatter Matrices are combinations of scatter plots, splitting the screen into several smaller subcharts. The aggregation model is the same as for basic scatter plots, i.e., extracting the records corresponding to either the biggest or the last shape per group. Given a sizing data column e, scatter matrices use the

CMA $Q_{max}(e; a, b, c, d, e)$, with a and b projecting to the x and y coordinates of the shapes in the individual scatter plots and c and d projecting to the layout dimensions k_u and k_v of the 2D grid of the matrix. Scatter matrices with equally sized marks use the last record per group, extracted using the CMA $Q_{max}(a; a, b, c, d)$.

The two aggregation variants can be combined to select appropriately aggregated records for both types of scatter matrices, i.e., with either variable or with equally sized marks. The corresponding CMA $Q_{max}(a, e; a, b, c, d, e)$ selects all records that either define the largest or the last shape per display unit.

Histograms are basic bar charts, displaying the value distribution of one data column. The computation of the histogram values is subject to the original query Q. The visual aggregation is the same as for basic bar charts, using a CMA $Q_{bar} = Q_{max}(b; a, b)$, with a projecting to the horizontal layout dimension of the basic bar chart and b determining the height of a bar.

Space-Filling Visualizations define a spatial order of all $w \cdot h$ pixels of the canvas according to a space-filling algorithm. Unlike scatter plots, the visualized dataset can have at most two data columns, one related the order of the pixels and the other determining the color of a pixel. Therefore, space-filling visualizations can be treated as bar charts with a height of $h = 1$ pixel and a width of $w' = w \cdot h$ pixels. For space-filling visualizations, the visual aggregation in each pixel can be simulated using the basic bar chart CMA $Q_{bar} = Q_{max}(b; a, b)$.

Gantt Charts have several lanes, displaying horizontal bars of variable size. New bars can start at any horizontal pixel position and have a size between one pixel and the number of pixels between the start position and the rightmost pixel of the canvas. Assuming records to be rendered from left to right, a bar starting in a specific pixel column will overplot previous bars that start or stretch into this pixel column. Gantt charts are approximated by selecting, for each lane and for each pixel column, the maximum, i.e., farthest stretching record, using the CMA $Q_{max}(s; a, b, c)$, with a projecting to the coarse horizontal layout dimension, i.e., determining the lanes, and b and c representing the start and end times of the intervals to be displayed in the Gantt chart. The auxiliary column s of Q_g determines the size of the bars and is computed similarly to that of stacked bar charts (cf. Section 5.3.2), using an additional subquery $Q_s = \pi_{a,b,c,s \leftarrow (c-b)}(Q)$ as input, instead of the original query Q.

Continuous Line Charts draw one line between every two pixel columns and one or more lines inside each pixel column. Overplotting occurs mainly inside each pixel column (cf. Section 4.5). As stated previously, a maximum aggregation and thus a CMA are not sufficient approximations for line charts. They require an M4 aggregation subquery with a corresponding correlation subquery as follows (also cf. Equation 4.5 in Section 4.5).

$$\begin{array}{rcl} A_{M4} & = & {}_{k}G_{k_G \leftarrow k, a_{min} \leftarrow min(a), a_{max} \leftarrow max(a), b_{min} \leftarrow min(b), b_{max} \leftarrow max(b)}(Q_g) \\ Q_{line} & = & \pi_{a,b}(Q_g \bowtie_\theta (A_{M4})) \\ \theta & \leftarrow & k = k_G \wedge (a = a_{min} \vee a = a_{max} \vee b = b_{min} \vee b = b_{max}) \end{array}$$

Discrete Line Charts are similar to continuous line charts, but use a smaller number of $w' = \lfloor w / w_{cell} \rfloor$ display units in the group key subquery Q_g. The visual aggregation for discrete line charts is computed using the value-preserving aggregation subquery Q_{line}, as defined above for continuous line charts.

Note that, for brevity, each CMA Q_{max} is defined only for the minimal set of data columns that is required to produce an appropriate visualization. In practice, additional non-aggregated data columns can be included in the query result, by adding them to the list of projected data columns. For instance, a specific scatter plot query $Q_{max}(t; t, v)$ for a specific time series $Q(id, t, v, cat)$ can be altered to $Q_{max}(t; t, v, id, cat)$, so that the categorical data columns id and cat can be used to enhance the visualization, e.g., defining the color or shape type of the drawn marks in the scatter plot.

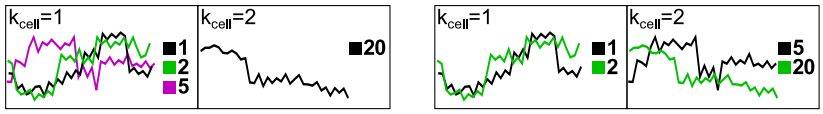
5.4. Conversion of Non-Numerical Data

The visual grouping of the considered data aggregation queries requires the input data to represent a *continuous* numerical data range. However, numbers in databases may also represent *ordinal* ranges, i.e., countable sets of distinct numbers. Therefore, such ordinal ranges and categorical data in general must be converted to continuous, i.e., sequential, numerical data before being reduced visually.

5.4.1. Sparse Numeric Identifiers

Continuous data, such as the time $t \in \mathbb{R}$ and value $v \in \mathbb{R}$ of a sensor signal can be directly used as input for VDDA, since visualizations of continuous data project each record to the likewise continuous 2D space of a visualization, i.e., using one or more linear projection functions (cf. Section 2.3, Equation 2.1).

Thereby, chart matrices use multiple linear projection functions, incorporated in the multidimensional grouping function (cf. Equation 5.1), not only for positioning individual records in the 2D space of each subchart, but also for distributing multiple series in the chart matrix. In this regard, non-sequential numerical data ranges may result in an unexpected distribution of the data subsets in the matrix.



(a) linear projection to grid cells (b) uniform distribution to grid cells

Figure 5.4.: Linear projection and uniform distribution of series identifiers.

Example 6. Consider for instance, the linear projection of an ordinal range, e.g., a specific set $S = \{1, 2, 5, 20\}$ of sensor *id*s, to the cells of a chart matrix. Figure 5.4a depicts a visualization of four series that are identified by an $id \in S$ and rendered to a conceptual 2×1 line chart matrix, where the mapping of series *id*s to matrix cells was computed using the following grouping function.

$$f(id) = \lfloor u \cdot (id - id_{min})/(id_{max} - id_{min}) \rfloor$$

This grouping function uses the range of *id*s of the given data to project each record to the corresponding coarse display units of the visualization, i.e., to the $u = 2$ cells of the matrix. Thereby, the series *id*s 1, 2, and 5 project to $f(id) = 0$ and the *id* 20 projects to $f(id) = 1$, resulting in a non-uniform distribution of series to matrix cells, as depicted in Figure 5.4a.

To better utilize the provided space in each canvas of the chart matrix, multiple datasets should be distributed uniformly to the matrix cells, as depicted in Figure 5.4b. This is achieved by considering *ordinal* ranges of numeric series identifiers as *categorical* data, whose projection to continuous ranges is described in the following section.

5.4.2. Categorical Data

Many datasets contain categorical data columns or a combination of categorical columns that represent series identifiers or identifiers for groups of series. Categorical data can have various types, including strings, such as the International Securities Identifying Number (ISIN) of a single stock market share or

the Market Identifier Code (MIC) of a specific stock market, but also numbers, such as the numeric *id* of a sensor, as described in the previous section.

Projecting categorical data to a sequence of display units, e.g., a stack or matrix of subcharts, requires to solve two problems. First, non-numerical data types must be converted to numerical data types, facilitating the execution of the arithmetic operations of the grouping function. Secondly, ordinal ranges with gaps, such as the set $S = \{1, 2, 5, 20\}$, must be converted to a sequence, e.g., $S' = \{1, 2, 3, 4\}$. In the following, any *sequence* S' is considered to have a fixed step size, since all considered visualizations use only linear functions to project data to layout dimensions. A detailed discussion of non-linear, e.g., logarithmic, projection functions is out of scope of this work.

The necessary conversion of categorical and ordinal data to sequential data can be achieved by iterating over the set of categories and assigning a running number to each unique category, e.g., to each ISIN, to each MIC, or to each series *id*. The resulting sequence of running numbers is compatible with VDDA, allowing for a uniform distribution of the sequential data to the display units of the visualization. Formally, given an input relation $Q(id, t, v)$, the conversion and a subsequent visual aggregation are conducted as follows.

1. Determine the set S of unique categories $id \in S$, having a length of $m = |S|$.

2. For each $id \in S$ assign a number $n \in \{1, 2, .., m\}$, resulting in a numbered category set $S'(id, n) = \{(id_1, 1), (id_2, 2), ..., (id_m, m)\}$.

3. Join the numbered category set S' with the input relation $Q(id, t, v)$, producing a numbered relation $Q_n(id, n, t, v)$.

4. Compute the visual aggregation for Q_n on the data column n instead of id, i.e., using a grouping function $f_g(n) = \lfloor c \cdot (n-1)/m \rfloor$ to uniformly distribute m series to c coarse display units, e.g., matrix cells.

5. Correlate, i.e., join, the aggregated data again with Q_n to obtain the original *id*s for each n in the aggregated data.

6. Project the original data columns id, t, v from the correlation result to restore the schema of the input relation.

The obtained result is a list of aggregated records (id, t, v), whose visual aggregation was computed jointly for groups of *id*s that project to the same matrix cell, based on their category number. In practice, the distribution to

Listing 5.4: Assigning sequence numbers to categorical data.

```
-- Q_rank, first variant assigns running number using a ranking function
SELECT DENSE_RANK() OVER(ORDER BY id) id, id oid, t, v FROM Q)

-- Q_rank, second variant assigns running number using row numbers
SELECT n id, Q.id oid, t, v FROM (
  SELECT ROW_NUMBER() OVER() n, id
  FROM (SELECT DISTINCT id FROM Q ORDER BY id)
) AS Q_n JOIN Q ON Q.id = Q_n.id
```

matrix cells can be adjusted by ordering the set S of unique categories, before computing their running numbers.

The described conversion can be implemented in SQL, either using the ranking function `DENSE_RANK`, or using the numbering function `ROW_NUMBER` on the set of `DISTINCT` categories, as shown in Listing 5.4. The used `ORDER BY` clauses determine on which data column the sequence numbers are defined, i.e., which data column is to be considered as set of unique categories. The listed queries can be used for VDDA as a substitute for the original query, i.e., as first subquery Q_{rank} on the original query Q. Note that the queries Q_{rank} rename the original *id* to *oid* and name the computed running numbers as *id*. The renaming facilitates that the subsequent VDDA subqueries retain their semantic, and must only be adjusted to use Q_{rank} instead of Q as input relation. Finally, after the new sequential *id*s are leveraged for the visual aggregation, the final subquery needs to restore the original values, by omitting the computed *id* column and renaming *oid* back to *id*.

The final query result will again have the same schema as the original query Q, and thus can be consumed transparently by the visualization client. Note that using the described query-level ranking requires the visualization to use an equivalent technique to uniformly distribute data subsets to the cells of a chart matrix. The result of a query-level visual aggregation of one group of series – assumed to be rendered jointly on a single canvas – is not guaranteed to correctly approximate or simulate the visual aggregation of other, different groupings of the data.

Essentially, for assigning running numbers at the query-level, the query rewriter must know how the visualization client orders categorical data, and reflect this ordering in the rewritten SQL queries, using appropriate `ORDER BY` and `COLLATE` clauses.

5.4.3. Temporal Arithmetic in SQL

In SQL databases, dates and timestamps cannot be aggregated visually per se, since most RDBMS do not support the basic arithmetic operations on temporal data that is required for processing the corresponding SQL data types in a VDDA query. In particular, the division of temporal data types may be restricted, and system-specific temporal conversion functions are required.

In many datasets, dates and timestamps represent continuous ranges, and thus cannot be processed using the conversion techniques for categorical data (cf. Section 5.4.2). Instead, a temporal conversion function must be used that preserves the continuity of the temporal data. Therefore, most SQL databases support a variety of built-in date, timestamp, and interval conversion functions.

The requirements for these functions are derived from the properties of the VDDA grouping function. A grouping function for a temporal data column

$$f(t) = \lfloor n_t \cdot (t - t_{min})/(t_{max} - t_{min}) \rfloor$$

incorporates the number of display units n_t, the timestamp t of each individual record and the minimum and maximum timestamps t_{min} and t_{max} of the acquired temporal range. The temporal values are used to define two temporal *intervals*. The *interval* $t - t_{min}$ is then divided by the *interval* $t_{max} - t_{min}$, obtaining a relative distance $t_r = (t - t_{min})/(t_{max} - t_{min})$ of the current record to the beginning of the temporal range, with $t_r \in [0, 1]$. As a result, the requirements for projecting a temporal data range to a layout dimension are the following.

Temporal Conversion Requirements

1. The database supports the computation of minimum and maximum timestamps.
2. Two timestamps or dates can be combined to represent their corresponding interval.
3. Intervals are dividable and their division yields a real number.

The minimum and maximum aggregation of dates and timestamps (Requirement 1) is implemented natively in most SQL databases. Moreover, many SQL databases, such as SAP HANA and SQL Server, support conversion of two SQL timestamps or dates to an interval length (Requirement 2), using the `datediff` or similar functions [11, p.141] that yield a real or integer number, depending on the implementation. If the `datediff` function yields an integer number

Listing 5.5: Boundary and group key subqueries using *datediff* function.

```
-- Q_b, boundary subquery using datediff function instead of subtraction
SELECT min(t) t1, 1.0 * datediff(millisecond, min(t), max(t)) dt FROM Q),

-- Q_g, group key subquery internally uses datediff and fully consumes dt
SELECT Q.*, floor($c * (1.0 * datediff(millisecond, t1, t))/dt) k FROM Q,Q_b
```

(as for SQL Server [66, p.77]), it can be converted to a real number through multiplication with 1.0 or using the commonly available `cast` function.[1]

In the following, the computation of a real number that represents the interval between two timestamps t_1 and t_2 is denoted as *interval function* $f_d(t_1, t_2) \rightarrow \mathbb{R}$. For instance, $f_d(t_1, t_2) =$ `1.0 * datediff(millisecond,$t1,$t2)` is an interval function that computes interval sizes with millisecond precision. Using such an interval function, a generalized temporal grouping function for timestamps or dates can be defined as follows.

$$f_g(t) = \lfloor c \cdot f_d(t_{min}, t)/f_d(t_{min}, t_{max}) \rfloor$$

In a VDDA query, the grouping function is integrated into the group key subquery Q_g, which depends only on the boundary subquery Q_b (cf. Listing 5.1). Using the described interval function and the temporal grouping function, the subqueries Q_b and Q_g can be modified to support temporal data, as exemplified in Listing 5.5.

Other SQL databases, such as PostgreSQL and MonetDB, allow converting timestamps to intervals by means of subtraction. Thereby, MonetDB represents the resulting intervals as integer numbers [22], instead of as specific interval data type, and thus only requires to convert the integer values to real numbers, e.g., using $f_d(t_1, t_2) = 1.0 \cdot (t_1 - t_2)$.

PostgreSQL supports the creation of `INTERVAL`s through subtraction, but their division is not supported [46, p.210]. Intervals must again be converted to numbers, e.g., using $f_d(t_1, t_2) =$ `EXTRACT(EPOCH FROM t1 - t2)`. The corresponding subqueries Q_b and Q_g are exemplified in Listing 5.6.

Formally, timestamp and interval conversion are only required for the group key subquery Q_g. For the boundary subquery, they are optional, since $dt = f_d(t_1, t_2)$ could also be computed directly by Q_g. The boundary subquery Q_b then needs to expose t_2 instead of dt. Other VDDA subqueries do not have to

[1] In practice, always convert a `datediff` result to a real number immediately. This avoids unexpected results when using it in a division operation.

Listing 5.6: Temporal boundary and group key subqueries in PostgreSQL.

```
-- Q_b, boundary subquery converting Postgres interval type to real number
SELECT min(t) t1, EXTRACT(EPOCH FROM max(t) - min(t)) dt FROM Q),

-- Q_g, group key subquery converting Postgres interval type to real number
SELECT Q.*, floor($w * EXTRACT(EPOCH FROM t - t1)/dt) k FROM Q,Q_b
```

be modified, since the interval values are consumed by Q_g and not exposed to the subsequent subqueries.

With the above extensions, VDDA is now capable of aggregating categorical data and moreover supports the conversion of temporal data types in SQL databases. This significantly broadens the range of scenarios, where VDDA can help to reduce the data volume, without compromising the quality of the visualization.

Summary

This chapter defined a visualization-driven data aggregation for the most common data visualizations. Therefore, this chapter first described how to decompose data visualizations into atomic display units and secondly generalized the building blocks of a visualization-driven data aggregation. Using this foundation, and following the exemplary description for formalizing the visual aggregation of a bar chart, this chapter defined specific VDDA queries of each considered type of visualization. The chapter concluded by giving practical advice how to convert non-numerical data types, including temporal data, to the required numerical and sequential data types.

6. Data Capacity of Common Visualizations

Chapters 4 and 5 defined how to conduct a visualization-driven data aggregation (VDDA) for the most common chart types and also described more generally how to define a corresponding data aggregation for any kind of visualization. Up to this point, VDDA was introduced as a purely spatial approach that avoids making any assumptions about how the user perceives the visualization. It does not guarantee well-perceivable visualizations of the data and may still result in undesirably high data volumes, in particular when processing multitudes of individual series.

The following chapter mitigates these shortcomings by defining how to spatially and perceptually limit the total number of series rendered in a data visualization. Therefore, the chapter first discusses applied perceptual limits, e.g., for categorical data, and secondly describes how multiple series spatially consume the provided white space of the visualization canvas. Formalizing this knowledge, this chapter extends VDDA towards a *spatio-perceptual* data reduction technique that allows a data visualization system (DVS) to determine how many and which of a multitude of series to display. By limiting not only the number of records per series, but now also the number of series per chart, the provided solution will ensure predictable data reduction rates for any VDDA query on any number of originally selected series with any number of records per series.

6.1. Problem Description and Solution Overview

In a basic chart and in each subchart of a chart matrix, multiple series may be rendered on a joint canvas. Thereby, each series may either be aggregated individually and visualized as overlaid rendering, or the displayed multitude of series may be aggregated jointly before rendering the combined data subset to the canvas (cf. Section 5.3.3). However, when visualizing large multitudes of series, neither of the two techniques ensures the perceptibility of each single

series, and both techniques may still result in undesirably large query results, as shown in the following examples.

Example 7. A user may intentionally or unintentionally choose to visualize the records of 1000 sensors in a line chart with a width of $w = 1000px$ and a height of $h = 500px$. The result of a related M4 query then contains data-reduced records for any of the 1000 sensors, i.e., at most $1000_{sensors} \cdot 4 \cdot 1000_w = 4M$ records, which constitute $96MB$ of uncompressed data to be transferred from the database, given a wire size of $3 \cdot 8byte$ per record.

The above example shows that a simple VDDA query still produces undesirably high data volumes for large numbers of individually rendered series. The problem can be partially mitigated by joint visual aggregation.

Example 8. In a scatter plot, the set of 1000 series may be aggregated jointly, resulting in a maximum data volume of $1000_w \cdot 500_h = 500k$ records ($12MB$), given that any of the 1000×500 pixels of the scatter plot represents at most one corresponding data point.

VDDA for scatter plots already supports aggregation over multiple series, whereby the resulting data volume is inherently limited by the number of display units (pixels) on the canvas (cf. Section 5.1). However, 1000 series is still a too large number of individual datasets to be visualized in a basic line chart or scatter plot. The perceptibility of individual sensor signals will be very low.

To improve the perceptibility of individual series, this work considers the DVS to explicitly limit the number of series using one or both of the following two approaches.

1. The DVS defines a limit to the total number of series to be visualized on the canvas of a single basic chart.

2. The DVS splits up the visualization into a chart matrix and thus lowers the resolution of the canvases of the subcharts.

Example 9. For Option 1, a limit of 10 series effectively reduces the acquired data volume for the line chart to $10_{sensors} \cdot 4 \cdot 1000_w = 40k$ records ($960kB$). Similarly, in a scatter plot, a limited number of series usually consumes less, e.g., only 5%, of the available white space, so that the acquired data volume is only $1000_w \cdot 500_h \cdot 0.05 = 25k$ records ($600kB$).

Example 10. For Option 2, given a line chart matrix of 10×10 cells spanning a total of 1000×500 pixels, the 1000 originally acquired series can be

distributed to 100 smaller line charts, each having a size of 100×50 pixels. A corresponding VDDA query produces $1000_{sensors} \cdot 4 \cdot 100_w = 400k$ records ($9.6MB$), representing the 1000 visually aggregated subsets that are required for correctly rendering 10 series in each subchart of the matrix.

Example 11. To further reduce data volumes and increase perceptibility of individual series, a series limit (Option 1) can be used for the chart matrix (Option 2). For instance, the DVS may consider displaying not more than 5 series in the small 100-pixel-wide line charts of the aforementioned chart matrix, thereby effectively reducing the data volume to $5_{series} \cdot 100_{cells} \cdot 4 \cdot 100_w = 200k$ records ($4.8MB$).

The exemplified approaches are seemingly simple, but they require the DVS to conduct a more complex reasoning, i.e., answering the following questions.

1. How many series are to be visualized, i.e., are contained in the original query result?
2. Given the number of series and the chart type, what is the best size for each subchart?
3. Given a set of display properties, how small may a subchart become?
4. Independent of the chart size, how many series should at most be rendered in each subchart?

Thereby, in particular Questions 2 and 4 cannot be answered separately but must be considered as a joint problem, i.e., to split up the canvas into a perceptually optimal chart matrix. But defining an appropriate size of the subcharts with a corresponding series limit is a difficult problem, having potentially different solutions for the various chart types. The underlying research question is the following.

Research Question. *What is the spatio-perceptual correlation between the size of the canvas and the number of series to be displayed in a data visualization?*

Solving this question requires a detailed analysis of how the rendered data consumes the available white space of a visualization, i.e., how varying the width and height of the visualization canvas affects the overplotting behavior of the rendered series. Thereby, a solution for basic charts will also serve as a solution for chart matrices, as those are mere compositions of multiple

basic charts. The presented solution to the described problem is based on the following three approaches.

1. The DVS always defines a reasonable *minimal chart size* to avoid low perceptibility of too small basic charts.
2. The DVS defines a *size-independent perceptual series limit* for any basic chart, to ensure perceptibility of individual data categories.
3. The DVS limits the number of series to a *size-dependent spatial series capacity* of a basic chart, aiming to mitigate excessive overplotting of overlaid series in smaller charts.

The following sections discuss this solution in detail, first motivating perceptual limits for the chart size, then defining a size-independent perceptual *series limit*, and eventually introducing a set of size-dependent *series capacity* functions that model the white-space consumption of the considered chart types.

6.2. Perceptual Limits

In common data visualizations, most pixels represent white space on the canvas rather than data values. Thereby, the contrast between white space and data-representing pixels is an important measure that determines the perceptibility of the rendered signal. If the canvas is too small, there may not be enough white space between the data points, resulting in a low perceptibility of the rendered data. Rendering multiple series on the same canvas will likewise consume the available white space and moreover overplot previously rendered data subsets, which also reduces perceptibility of the visualization.

Therefore, a perception-aware DVS must define the size, i.e., all parameters of a visualization, such that the user is capable of determining a maximum of features of all visualized data subsets. Optimally, this would require an expensive simulation of the process of human perception using a computational model [84], which is out of scope of this work. More practically, the DVS may define a set of perceptual limits, whose application can significantly improve the perceptibility of a visualization, without requiring additional expensive computations on the entire data set. Thereby, this work considers the following perceptual limits.

Minimal Chart Size. The minimal chart size defines the smallest possible chart size that still allows the DVS to produce a perceptible rendering of the

data. The minimal chart size is represented by the lower bounds w_{min} and h_{min} for the width w and height h of a basic chart or the subcharts in a chart matrix.

Perceptual Series Limit. The perceptual series limit m_P on the number of jointly rendered series m defines how many series a visualization should display at most, independent of the spatial properties of the visualization.

Note that this work does not consider specific upper bounds on the spatial parameters w and h, since display resolutions of stationary screens and mobile devices are increasing continuously. It is also not useful to define a specific lower bound on the number of jointly rendered series m, since a database query may yield only a single series ($m = 1$) or no result at all ($m = 0$).

The following subsections, briefly motivate the specific perceptual limits used in this work, defining and discussing specific values for w_{min}, h_{min}, and m_P, independent of the type of visualization. Note that, in practice, these values may be adjusted by the DVS, based on the chart type and a variety of visualization-related parameters, whose comprehensive analysis is out of scope of this work.

6.2.1. Choosing a Minimal Chart Size

Theoretically, w_{min} and h_{min} can be very small and still produce a reasonable basic chart of a single series or even of multiple series. For instance, a DVS may define $w_{min} = 5$ and $h_{min} = 3$, as illustrated in Figure 6.1. The depicted charts may theoretically still represent a reasonable visualization of the data at the pixel level, as illustrated by the 8:1-magnified images in the figure, but on a high-resolution screen they will become physically very small, as illustrated by the 2:1 and 1:1 versions of the images in the figure.

Physically small charts generally suffer from reduced perceptibility, particularly when rendering real-world data, rather than an artificial dataset, as was used in Figure 6.1. Therefore, this work considers $w_{min} = 20$ and $h_{min} = 10$ as more reasonable default values for the smallest possible chart size, to avoid low perceptibility of physically small basic charts. The effect of using these higher default values over the described smaller values is illustrated in Figure 6.2, which shows a 1:1 screenshot of a line chart matrix next to a 3:1-magnified version of the screenshot for each parametrization. Thereby, Figure 6.2a illustrates that a line chart matrix with 5×3 pixel subcharts still allows the viewer to perceive rough shapes of the signals at the 3:1-magnified pixel level, but that, at their native 1:1 resolution, the reduced physical size of the subcharts

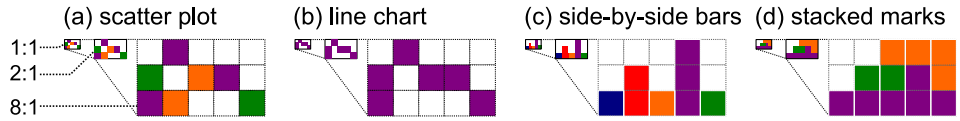


Figure 6.1.: Very small basic charts with 5×3 pixels.

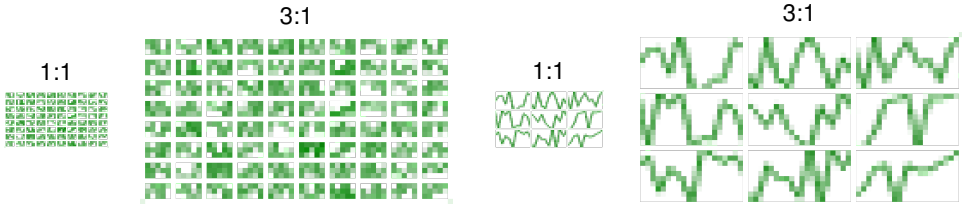


Figure 6.2.: Perceptibility of single series in very small line charts.

results in a significantly reduced perceptibility of each individual signal. In contrast, the 20×10 pixel subcharts in Figure 6.2b provide a larger physical size and a higher amount of pixels to display individual features of the data, resulting in a significantly improved perceptibility of the visualization.

The physical size of a visualization is one problem. The ability to correctly display particular features of the data on a limited pixel space is another. For instance, the data appears to be very dense in many of the 5×3 pixel subcharts, sometimes coloring all 15 pixels. In such overplotted charts, the user may interpret dense areas as areas of high variance and high frequency, as moreover illustrated in Figure 6.3b. However, the underlying data may not be as dense as it appears, which is only revealed by projecting the data to the larger 20×10 pixel charts, as depicted in Figure 6.3a.

For the depicted data subsets, a canvas size of 20×10 pixels provides enough space to not only show the existence of $1-10$ minima and maxima of the signal, but also a well-defined shape of the slopes left and right of each minimum and maximum. In comparison, the rendering of the signal to 5×3 pixels provides only a very rough approximation of the original signal.

Formally, the minimum size of the canvas must be chosen according to the minimum possible size of the corresponding marks, i.e., such that a desired number n_f of features of the data can be visualized. For a line chart of a continuous sensor signal, the following exemplary reasoning may be considered.

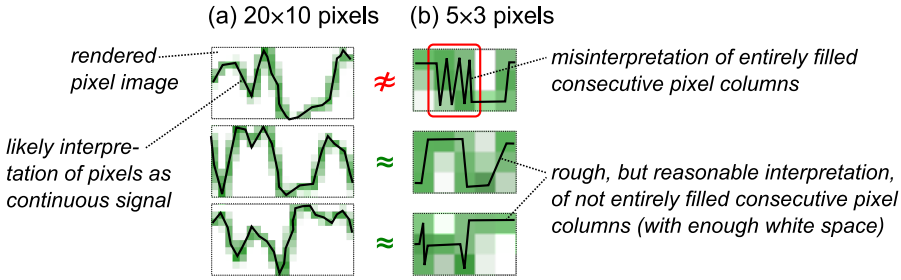


Figure 6.3.: Misinterpretation of dense areas in small line charts.

Example 12. In a line chart with a minimum line thickness of $w_{line} = 1$ pixel, a single perceptible feature of the data, e.g., a slope of the continuous sensor signal, requires a single line segment the span at least $w_{feat} = 2$ pixel columns. Given a width of the chart of $w_{min} = 5$ pixels, the number of displayable features is $n_f = w_{min}/w_{feat} = 2.5$. Alternatively, given a number of desired features of $n_f = 10$, the required minimum width of a line chart is $w_{min} = n_f \cdot w_{feat} = 20$ pixels.

The above example motivates the definition of the default width $w_{min} = 20$, whereby a corresponding default height of $h_{min} = w_{min}/2 = 10$ provides an aspect ratio of $2:1$ that roughly matches the aspect ratio of current display devices. A more comprehensive definition and evaluation of various parametrizations of w_{min} and h_{min} for each individual chart type are out of scope of this work. To retain the generality, the remainder of this section considers w_{min} and h_{min} as variables, to be set by the DVS depending on any determined spatial, perceptual, or physical property of the considered visualization or the connected display device.

6.2.2. Choosing a Maximum Number of Series

Many visualizations show several series jointly on one canvas. In this regard, larger charts can usually display a higher number of series per chart than smaller charts, i.e., with the number of series increasing proportionally to the width, height, or area of the chart. However, when considering the limits of human perception, not more than a few dozen series should be displayed in one single chart independent of its size, since the number of easily distinguishable colors, shapes, and line styles is very limited.

Thereby, varying the color is the most frequently used approach to distinguish different data categories, and a maximum of seven to twenty colors is

widely accepted and encoded in visualization libraries. For instance, the widely used D3 data visualization library [19] (d3js.org) provides the methods `d3.scale.category10()` and `d3.scale.category20()` to assign up to twenty colors to categorical data. Other sources, such as ColorBrewer [50] (colorbrewer2.org), suggest using even fewer colors, e.g., not more than seven, and up to twelve colors if the colored data subsets are spatially separated. Similar values are confirmed by other studies [51] and are also inherent in data visualization tools like Tableau[1], usually supporting up to twenty colors for representing categorical data.

The variation of colors can be combined with the variation of shapes and line styles, so that the theoretical total number of distinguishable categories can be higher for some types of visualizations. Nevertheless, this work regards displaying more than twenty series in any of the supported basic chart types as impractical and consequently uses a default value of $m_P = 20$ for the size-independent perceptual *series limit*. Again note that m_P is defined as a variable in the remainder of this section, to retain the generality of the presented approach.

6.3. Spatio-Perceptual Capacity

The perceptual limits, defined in Section 6.2, constrain the data capacity of a visualization in general, independent of the actual chart size. Nevertheless, the actual data capacity of a specific visualization also depends on its specific pixel width and pixel height.

For a given width, height, and type of visualization, the DVS must automatically determine the data capacity, to subsequently decide how many and which of the multitude of selected series to display and thus acquire from the database. To automate this process, this work provides a formalization of the data capacity of a visualization, which was previously defined only informally as *visual scalability* [32] (cf. Section 2.8.3). This visual scalability is now formalized using the following two *visual capacity* measures.

Definition 7. Base Capacity. The base capacity $n_C = f(w, h)$ is the maximum number of data points per series that a visualization of $w \times h$ pixels can display unambiguously.

Definition 8. Series Capacity. The series capacity $m_C = f(w, h)$ is the maximum number m_C of series that a visualization of $w \times h$ pixels can display comprehensively, without excessive overplotting of individual series.

[1] Evaluated version: Tableau Desktop 8.1 (tableausoftware.com).

A discussion of the base capacity n_C is provided in Chapter 5 that defines how a visualization is decomposed into its n_C *display units*. In the following, this chapter focuses on the series capacity m_C.

6.3.1. Series Capacity Functions

The series capacity of a specific chart type is first determined by the size-independent perceptual series limit m_P, and secondly by how the available white space is consumed by the graphical marks of the chart type. Thereby, for visualizations that are designated to display data subsets as overlaid renderings, such as scatter plots and line charts, a proposed series capacity function can only provide an approximation of how white-space consumption generally affects the series capacity of the chart type. The actual series capacity of a specific visualization always depends on the data distribution of the visualized multitude of series and would theoretically require an upfront simulation of the rendering process to determine the actual amount of occupied pixels per series. However, upfront rendering of big data in the back-end for mere testing and evaluation purposes, i.e., any upfront reasoning on the actual data distribution of the underlying data, is out of scope of this work. For visualizations that do not overlay multiple series, such as most forms of bar charts and tables, the series capacity can be formalized precisely and in general.

For each chart type, two specific capacity functions $n_C = f(w, h)$ and $m_C = f(w, h)$ can be defined to model the data capacity of the chart type. Thereby, the base capacity n_C scales linearly with the number of display units of each chart type (cf. Section 5.1, Table 5.1 on page 97). The series capacity m_C has a more complex behavior that requires analyzing how a single series consumes the available white space vertically and horizontally.

Thereby, the total amount of white space of an empty canvas is $w \cdot h$. Given that the size of the visual marks is constant, e.g., the thickness of a line does not change for different chart sizes, the marks will be relatively larger and occupy more white space on smaller charts and they will be relatively smaller and occupy less white space on larger charts. This observation is illustrated in Figures 6.4 and 6.5, showing how vertical and horizontal scaling affects white space consumption in a line chart and in a scatter plot, whereby larger charts provide additional white space to be used for rendering additional series. Based on this observation, the capacity functions for the considered chart types are defined as follows.

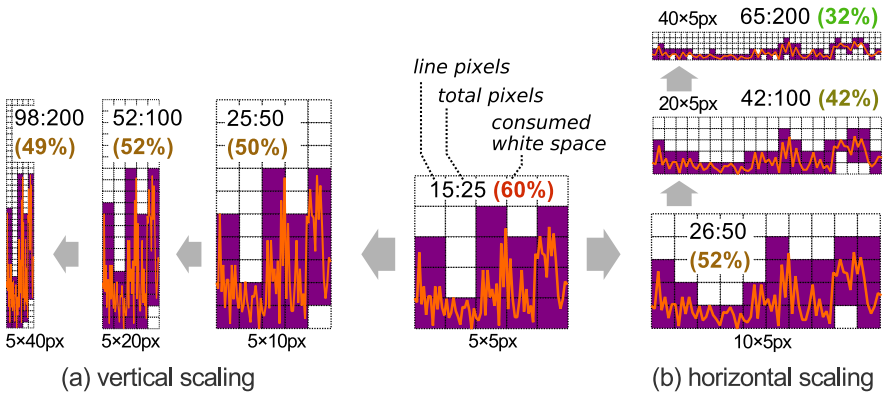


Figure 6.4.: White space consumption in line charts.

Line Charts require only up to $n_C = 4 \cdot w$ records per series for a pixel-perfect visualization of the raw data (cf. Section 4.5.2). Increasing the *height* of the chart only slightly increases the amount of white space (cf. Figure 6.4a), while increasing the *width* of the chart does significantly increase the amount of white space (cf. Figure 6.4b). Consequently, the maximum number of series m_C, displayed in a single chart, can scale horizontally with the width w of the chart. Thereby, one approach to approximate the scalability of line charts is to allow for one additional series per w_{min} pixels. Incorporating the perceptual series limit m_P, the horizontal scaling of the series capacity of a line chart can be approximated as $m_C = \min(m_P, w/w_{min})$.

An alternative series capacity for line charts is $m_C = \min(m_P, \sqrt{w})$, which allows for more than one series even in very small charts. This can be useful for datasets of low variance that are less prone to overplotting.

Scatter Plots can display at most $n_C = w \cdot h$ records per series. Since the marks are not graphically connected as in line charts, scatter plots scale equally well in both spatial dimensions (compare Figure 6.5a and Figure 6.5b). Consequently, the maximum number of series m_C for scatter plots is equally determined by the width w and height h, and again limited by m_P. The series capacity of a scatter plot can be approximated as $m_C = \min(m_P, \sqrt{w \cdot h})$.

An alternative series capacity for scatter plots is $m_C = min(m_P, A/A_{min})$, with $A = w \cdot h$ and $A_{min} = w_{min} \cdot h_{min}$, which further constrains the number of series in smaller charts, given reasonably high values for w_{min} and h_{min}. This

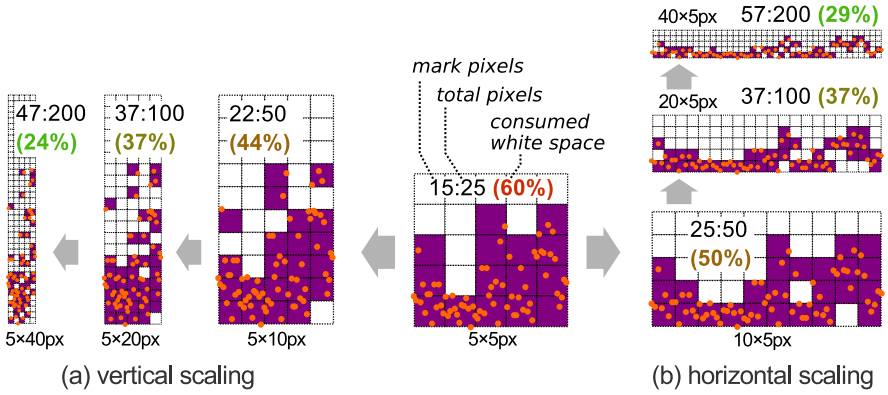


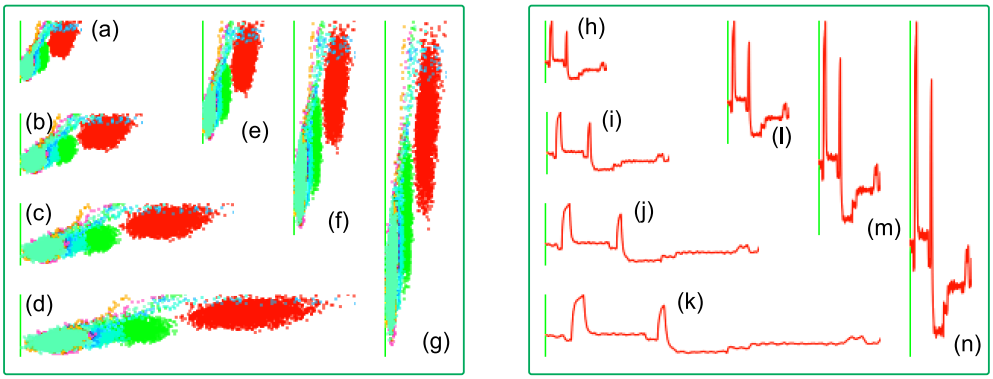
Figure 6.5.: White space consumption in scatter plots.

can be useful for data subsets that spread out over the entire canvas and thus will consume a lot of white space in small charts.

Figure 6.6 depicts additional examples to demonstrate the described behavior of the white space consumption. The figure shows a scatter plot with multiple series and a line chart with a single series, depicted at various sizes. Thereby, any wider or higher scatter plot and any wider line chart reveals additional details of the data on the gained white space. Line charts of increased height do not provide additional details because most of the additional white space is consumed by the rescaled lines.

Bar Charts can display up to w series, using one bar per series with $w_{bar} = 1px$. They can also display up to $n_C = w$ records, when rendering only a single series. A bar always spans the same fraction of the height of the chart, independent of the actual height of the visualization. Therefore, the height of the chart does neither influence the number of series m_C nor the number of records n_C per series. Consequently, the series capacity of a bar chart is defined as $m_C = min(m_P, w)$.

Stacked Bars can theoretically display up to $n_C = w$ records of each of up to h different series, e.g., if each stacked bar in a pixel column is exactly 1 pixel high. When considering real-world data, many of the stacked bars will consume more than one consecutive pixel of a pixel column and thus reduce the maximum number of visible series per pixel column. However, considering a complete stacked bars chart with multiple pixel columns, each series usually



Resizing a scatter plot (a) horizontally (b-d) or vertically (e-g) reveals additional details. Resizing a line chart (h) horizontally (i-k) reveals details, but resizing vertically (l-n) does not.

Figure 6.6.: Line charts vs. scatter plot scaling

has smaller bars in one column and bigger bars in other columns. A viewer will still gain insight into each series, even when using $m_C = h$ series (as used in Figure 6.1d), so that the series capacity of stacked bars charts can be defined as $m_C = \min(m_P, h)$.

The above discussion covered line charts, scatter plots, and bar charts. The other basic chart types and thus their capacity functions can be derived from one of the above types as follows.

Side-by-Side Bars are regular bar charts of one or multiple series and thus have a base capacity of $n_C = w$ and a series capacity of $m_C = \min(m_P, w)$.

Aligned Bars, Measure Bars, and Histograms are specialized bar charts, displaying only one single series. Their base capacity is $n_C = w$ and their series capacity is $m_C = 1$.

Space-Filling Visualizations are formally bar charts where each pixel represents a specific record. Consequently, the resulting base capacity is $n_C = w \cdot h$ and the series capacity is $m_C = min(m_P, w \cdot h)$.

Circle Charts are low-resolution scatter plots with $w' = w/w_{sub}$ composite pixel columns, each having a width of w_{sub} pixels. In each composite column, overplotting occurs only vertically over the h pixel rows of the column. Similar

to scatter plots, the base capacity is $n_C = w' \cdot h$ and the series capacity is $m_C = \min(m_P, \sqrt{w' \cdot h})$.

Discrete Line Charts are low-resolution line charts with $w' = w/w_{sub}$ composite pixel columns. Their base capacity is $n_C = w'$ and their series capacity is $m_C = \min(m_P, w'/w_{min})$.[2]

Given the above capacity functions for basic charts, the approach can now be extended to chart matrices.

Chart Matrices are visualizations of $W \times H$ screen pixels, composed of $u \times v$ subcharts, laid out in $u = \lfloor W/w \rfloor$ matrix columns and $v = \lfloor H/h \rfloor$ matrix rows. Each subchart displays up to m_C series. Consequently, the total *matrix capacity* of any type of chart matrix is defined as follows.

$$M = u \cdot v \cdot m_C \tag{6.1}$$

Gantt Charts stack multiple series in v lanes. They are similar to aligned bars matrices with $u = 1$ matrix columns, i.e., they display $m_C = 1$ single series in each of the v lanes with up to $n_C = w$ records per series. Consequently, they have a matrix capacity of $M = v$ series.

[2] The actual scalability of a discrete line chart depends on how the DVS constrains its usage. Focusing on big data, this work assumes that a discrete line chart with w' cells is used to display a larger number of $w' \gg u$ records, so that records need to be aggregated visually in the limited number of cells – even though this may be considered an uncommon usage of discrete line charts. Thereby, all records of a cell are rounded to the same x-coordinate value, which causes heavy overplotting inside the middle pixel column of the cell, while the remaining margin columns gain a lot of white space. In fact, the consecutive margin columns between two middle columns display at most one inter-cell connection line per series, while all actual data points and their inner-cell connection lines reside inside the middle column. From a big-data point of view, a discrete line chart with w' cells is the same as a basic line chart with $w = w'$ consecutive pixel columns, since the information provided by the $w - w'$ margin columns is negligible, only representing a small sample of the big data. Even though the margin columns provide a detailed view on this sample, it is still a *small* and *random* sample that provides little to no additional information to the viewer.

In non-big-data scenarios with $n \leq w'$ underlying data points, displaying up to w' data points in a discrete line chart can be very beneficial. By using enough margin space a discrete line chart ensures that all line segments are displayed with sufficient detail, i.e., using at least w_{sub} pixels. This allows the viewer to perceive the slopes of all line segments easily.

chart type category		*chart type*	m_C *(series per chart)*	n_C *(records per series)*
2D plots		scatter plot	$\min(m_P, \sqrt{w \cdot h})$ $\min(m_P, w \cdot h / A_{min})$	$w \cdot h$
		circle chart	$\min(m_P, \sqrt{h \cdot w / w_{sub}})$ $\min(m_P, h \cdot (w/w_{sub}) / A_{min})$	w / w_{sub}
		Gantt chart	1 (per lane)	w
1D charts (line charts)		line chart	$\min(m_P, w/w_{min})$ $\min(m_P, \sqrt{w})$	$4 \cdot w$
		discrete line chart	$\min(m_P, (w/w_{sub})/w_{min})$ $\min(m_P, \sqrt{w/w_{sub}})$	$4 \cdot w / w_{sub}$
1D charts (bar charts)		bar chart	$\min(m_P, w)$	w
		side-by-side bars	$\min(m_P, w)$	w
		stacked bars	$\min(m_P, h)$	w
		aligned bars	1	w
		histograms	1	w
		measure bars	1	w
		space-filling vis.	$\min(m_P, w \cdot h)$	$w \cdot h$
2D grids		tables	1	1
		heat maps	1	1
		highlight tables	1	1
chart matrix		any	$M = u \cdot v \cdot m_C$	$u \cdot v \cdot m_C \cdot n_C$

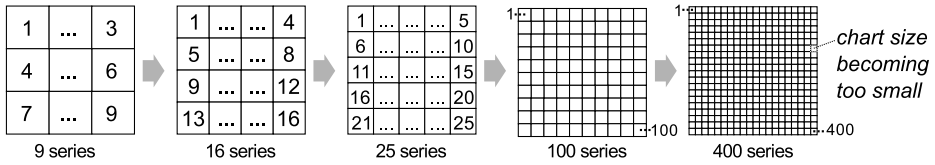
Table 6.1.: Series capacity m_C and base capacity n_C of common charts.

Text Tables, Heat Maps, and Highlight Tables are matrices with $u \times v$ cells, displaying exactly one value in each cell and thus not more than $m_C = 1$ series per cell. Consequently, their matrix capacity is $M = u \cdot v$.

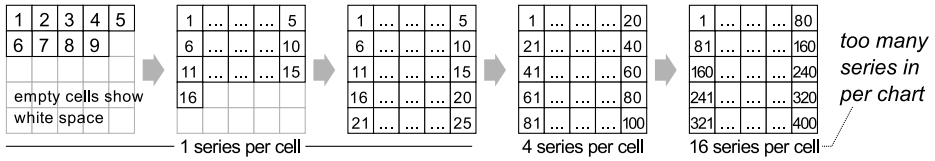
Table 6.1 lists all data capacity functions developed in this work and thereby concludes the definition of a base capacity n_C, a series capacity m_C, and a matrix capacity M for all considered basic and composite chart types (cf. Section 2.8.2).

6.3.2. Spatio-Perceptual Scaling of Chart Matrices

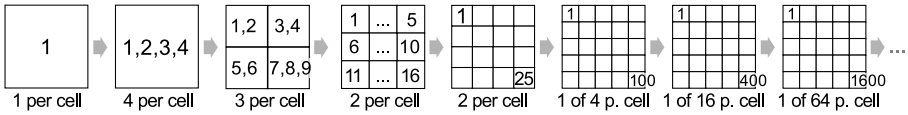
Given the developed capacity functions for chart matrices, a DVS can maximize the number m_C of series per chart, depending on the width w and height h of the subcharts, and thus minimize the number c_{uv} of cells in the chart matrix, while respecting the perceptual constraints $m_C \leq m_P$, $w \geq w_{min}$, and



(a) Number of cells is equal to the number of series. Cell size is adjusted to fill screen.



(b) Number and size of cells is constant. If $m > u*v$, multiple series start sharing cells.



(c) Number of cells, size of cells, and series per cell derived from perceptive limit m_C.

Figure 6.7.: Naive vs. perceptual scaling of chart matrices.

$h \geq h_{min}$. This approach is superior to naive scaling techniques, as discussed in the following.

An unconstrained scaling of the size and number of matrix cells, e.g., according to the number m of originally selected series, may result in chart matrices of low perceptibility. Figure 6.7 illustrates the drawbacks of such naive scaling approaches in comparison to the proposed scaling technique.

Thereby, Figure 6.7a depicts how solely scaling the number of cells, and adjusting its size accordingly, affects the perceptibility of the chart matrix. When visualizing more than a few dozen series on a small canvas or a few hundred on a larger canvas, the matrix cells will quickly become too small to present a meaningful subset of values for each individual series.

Figure 6.7b considers a second scaling approach using a fixed number of cells with a fixed cell size. If the number of visualized series m exceeds the maximum number of cells c_{uv}, multiple series must now share one cell. The matrix cells will suffer from heavy overplotting if $m \gg c_{uv}$. The first approach allows for bigger and thus better subcharts when visualizing a small number of series, while the second approach works better for visualizing a large number of series, allowing for multiple series to be visualized jointly in one cell.

The scaling technique, developed in this work, combines these two approaches by similarly scaling the number c_{uv} of cells for an increasing number m of selected series and by moreover incorporating the series capacity m_C of the subcharts, to provide perceptible renderings of each individual series. The approach is illustrated in Figure 6.7c. In contrast to the naive scaling, this perception-aware scaling aims to provide the following features.

1. Allowing for larger subcharts, by jointly visualizing multiple series on the subchart canvas.
2. Limiting the number of series in a subchart, depending on its size, to ensure perceptibility of individual series.
3. Limiting the size of the subcharts to a predefined minimum, to ensure general perceptibility on physical displays.
4. Limiting the total number of series in a chart matrix, by combining the series and size limits above.

Depending on which parameters of a visualization are predefined for a specific visualization scenario, the DVS can use the scaling functions $m_C = f(w, h)$ and $M = u \cdot v \cdot m_C$ to determine the unknown parameters, as described in the following.

6.4. Matrix Scaling Applications

The developed capacity functions allow for the following two kinds of optimization and automation.

Automatic Chart Matrix Configuration. A visualization system is advised to display m series. Knowing the series capacity $m_C = f(w, h)$ for the desired chart type, it can automatically determine the number of cells of a chart matrix to display all m series comprehensively, i.e., with maximum detail for the given capacity and number of series.

Pruning. A visualization system is advised to display m series in a $u \times v$ chart matrix. With w and h predefined by the application, the system can determine the specific series capacity m_C and the corresponding matrix capacity M. Given that $m > M$, the system can then select m_C series per cell and prune any extent series to speed up data acquisition and subsequent data processing.

6.4.1. Automatic Chart Matrix Configuration

Existing works on automatic charting focus on automatically choosing the chart type [77, 83], but neglect to define how to configure the chosen charts. However, an appropriate configuration is particularly important for large and high-dimensional datasets that are visualized in chart matrices.

In this regard, an automatic configuration of the chart matrix is useful when the acquired number of series m does not exceed the total number of displayable series of the largest considered matrix, i.e., with $w = w_{min}$ and $h = h_{min}$. The DVS can then, for a given available screen space of $W \times H$ pixels, determine the optimal cell width w and height h and thus the optimal number of matrix columns $u = W/w$ and rows $v = H/h$, i.e., it can solve w and h for $m = u \cdot v \cdot m_C = u \cdot v \cdot f(w, h)$ (cf. Equation 6.1).

The solution is non-trivial, since w and h and thus u and v in turn determine $m_C = f(w, h)$. To obtain a single solution, rather than a large solution space, the DVS must first define a fixed aspect ratio $q = w/h$ of the subcharts, e.g., using $q = w_{min}/h_{min}$. Given a predefined aspect ratio q, the problem can be rewritten to solve $c = u \cdot v = m/m_C$ for the number of matrix cells c as single solution variable. Therefore, the area of a subchart $A = w \cdot h$ can be defined as a function of c to likewise denote the unknown variables w and h as a function of c.

$$\begin{aligned} A &= f(c) = W \cdot H/c \\ w &= f(c) = \sqrt{q \cdot A} \\ h &= f(c) = \sqrt{A/q} \end{aligned} \tag{6.2}$$

The problem is summarized as follows.

Given	W the width of the available screen space, H the height of the available screen space, w_{min} the minimal width of a subchart, h_{min} the minimal height of a subchart, q the aspect ratio of a subchart, m_P the perceptual series limit, $m_C = f(w, h)$ the series capacity function for a specific chart type, m the number of series to be visualized,
Solve	$c = m/m_C$, $c = u \cdot v$, $u = W/w$, $v = H/h$, $A = W \cdot H/c$, $w = \sqrt{q \cdot A}$, $h = \sqrt{A/q}$,
With	$c > 0$, $w > 0$, $h > 0$, $q > 0$,
For	c the number of cells in the matrix.

Note that the above problem statement omits any rounding or truncation of decimal numbers to integers, since this would obstruct the general solvability of the problem.

The given equations yield different solutions for each of the considered chart types. Therefore, the series capacity can be denoted as $m_C = min(m_P, m_U)$, with the *unconstrained series capacity* m_U defining the series capacity of a subchart that is not constrained by the perceptual series limit m_P. The sought number of matrix cells c is then defined by the following *matrix scaling function*.

$$\begin{aligned} c = &\quad m/m_C = m/\min(m_p, m_U) \\ c = &\begin{cases} m/m_P & \text{if } m_P \leq m_U \\ m/m_U & \text{else} \end{cases} \end{aligned} \tag{6.3}$$

The equation is solved by defining m_U using only the given variables. Therefore, based on the proposed series capacity functions for the considered chart types (cf. Table 6.2), chart-specific solutions for m_U are the following.

Scatter Plot matrices have $c = m/\min(m_P, \sqrt{w \cdot h})$ cells. Given $m_U = \sqrt{w \cdot h}$, $c = m/m_U$, and Equations 6.2, the unconstrained capacity of a single scatter plot can be defined as $m_U = \sqrt{A} = \sqrt{W \cdot H/c}$ and thus as $m_U = W \cdot H/m$, independent of the unknown variables w, h, and c.

For the alternative scatter plot capacity function $m_C = \min(m_P, A/A_{min})$, the unconstrained capacity is $m_U = m$, since a larger subchart can display the same number of series as several smaller subcharts on the same screen space.

Line Chart matrices have $c = m/\min(m_p, w/w_{min})$ cells. Using Equations 6.2 to define the width of a subchart as $w = \sqrt{q \cdot A} = \sqrt{q \cdot W \cdot H/c}$, the unconstrained series capacity of a subchart can be defined as $m_U = (q \cdot W \cdot H/w_{min}^2)/m$.

For the alternative line chart capacity function $m_C = \min(m_p, \sqrt{w})$, given $w = \sqrt{q \cdot W \cdot H/c}$ and $m_U = m/c$, the unconstrained capacity is $m_U = \sqrt[3]{q \cdot W \cdot H/m}$.

Bar Chart matrices have $c = m/\min(m_p, w)$ cells. Given Equations 6.2 and $m_U = m/c$ the unconstrained series capacity of each subchart is $m_U = q \cdot W \cdot H/m$.

Stacked Bars matrices have $c = m/\min(m_p, h)$ cells. Given Equations 6.2 and $m_U = m/c$, the unconstrained series capacity of each subchart is $m_U = (W \cdot H/q)/m$.

	chart type	m_C	m_U
	scatter plot	$\min(m_p, \sqrt{w \cdot h})$ $\min(m_p, A/A_{min})$	$W \cdot H/m$ m
	line chart	$\min(m_p, w/w_{min})$ $\min(m_p, \sqrt{w})$	$(W \cdot H/w_{min}^2)/m$ $\sqrt[3]{q \cdot W \cdot H/m}$
 	bar chart side-by-side bars	$\min(m_p, w)$	$q \cdot W \cdot H/m$
	stacked bars	$\min(m_p, h)$	$(W \cdot H/q)/m$
  	aligned bars histograms measure bars	1	1
	space-filling vis.	$\min(m_p, w \cdot h)$	m
	discrete line chart	$\min(m_p, (w/w_{sub})/w_{min})$	$\frac{q \cdot W \cdot H}{w_{min}^2 \cdot w_{sub}^2}/m$
	circle chart	$\min(m_p, \sqrt{h \cdot w/w_{sub}})$	$(W \cdot H/w_{sub})/m$
	Gantt chart	1 (per lane)	1
  	tables heat maps highlight tables	1	1

Table 6.2.: Constrained and unconstrained series capacities.

Defining m_U for the remaining chart types is either trivial, as for aligned bars and tables with $m_C = m_U = 1$ and thus $c = m$, or they are a variant of the aforementioned chart types with one of the known variables multiplied by a given factor. Table 6.2 shows for each considered chart type the series capacity m_C next to the corresponding unconstrained series capacity m_U.

The derived specific definitions of m_U complement the generic matrix scaling function (cf. Equation 6.3). The DVS can now for any number m of selected series, and given W, H, w_{min}, h_{min}, and q, derive the optimal number of matrix cells $c = m/\min(m_P, m_U)$, as a decimal number with $c > 0$.

In practice, the minimum number of cells in a matrix is $c = 1$, so that the DVS introduces an auxiliary variable $c' = \max(1, c)$. The final discrete number of cells and the corresponding discrete chart matrix parameters are then obtained as follows.

First, the width and height of the matrix cells are computed using Equations 6.2. Thereby, c is substituted by c', and the result is moreover constrained by

the pixel size of the matrix and the minimal chart size.

$$\begin{aligned} w_s &= \max(w_{min}, \min(W, \sqrt{q_{wh} \cdot W \cdot H/c'})) \\ h_s &= \max(h_{min}, \min(H, \sqrt{(W \cdot H/c')/q_{wh}})) \end{aligned} \tag{6.4}$$

Subsequently, the discrete numbers of matrix columns u and matrix rows v are derived from the computed w_s and h_s.

$$\begin{aligned} u &= \lfloor W/w_s \rceil \\ v &= \lfloor H/h_s \rceil \end{aligned} \tag{6.5}$$

Thereafter, the discrete pixel width w_c, the discrete pixel height h_c, and the final discrete number c_{uv} of matrix cells are computed as follows.

$$\begin{aligned} w_c &= \lfloor W/u \rfloor \\ h_c &= \lfloor H/v \rfloor \\ c_{uv} &= u \cdot v \end{aligned} \tag{6.6}$$

Note that c_{uv} may slightly differ from the initially computed number of cells c obtained from the matrix scaling function, since Equations 6.6 and 6.5 do introduce discretization errors. The described analytical solution for c is only an approximation but solvable in general. The approximated problem of intersecting the matrix into c *discrete* and *equally* sized cells cannot be solved for all possible combinations of W and H, i.e., such that any of the $W \cdot H$ discrete pixels are included in one of the resulting cells. Depending on the possible factorizations of $c = u \cdot v$, only a very limited set of discrete matrix configurations exists that comprise all $W \cdot H$ pixels in the c cells.

6.4.2. Matrix Configuration for VDDA

The computation of the matrix parameters w_c, h_c, and c_{uv} requires the DVS to determine the originally selected number of series m. Therefore, instead of fetching all records of all series from the database, the system can issue a fast counting query, such as the following SQL query.

```
WITH Q AS (SELECT id,t,v FROM sensors) -- original query
SELECT count(distinct id) m FROM Q     -- number of unique series
```

To speed up counting queries on series identifiers, the database must provide appropriate indexing, e.g., having defined an index on `sensors(id)`.

The DVS can quickly count the number of originally selected series and subsequently derive an optimal chart matrix configuration as described in Section

6.4.1. The derived configuration can then be used for VDDA, i.e., the DVS can incorporate the matrix parameters as scalar values in a single-row subquery $Q_p(w_c, h_c, c_{uv})$ of a VDDA query.

Theoretically, the aforementioned counting query and thus Q_p can be derived inline, i.e., without previously issuing a separate counting query and thus without causing any round trip to the database. The corresponding implementations of Equations 6.3, 6.4, and 6.5 as SQL queries can be found in Appendix B. In the following, only the final subquery Q_P is discussed and included in the corresponding listings.

Listing 6.1 exemplarily shows the subqueries of a VDDA query that incorporates the set of matrix parameters to compute the visual aggregation for each subchart of a $W \times H$ scatter plot matrix. The query includes all relevant subqueries of a common VDDA query, the original query Q, a boundary subquery Q_b that determines the value range of the visually relevant data dimensions, a group key subquery Q_g that groups mutually overplotting records, a visual aggregation subquery A_{scat} that computes the last overplotting values, a correlation subquery Q_{scat} that assigns original records to the aggregated values, and a final projection subquery that maps the aggregated data to the original schema of Q for facilitating a transparent data consumption at the client side. The matrix parameters w_c, h_c, and c_{uv} are used by Q_g for clustering the data into one of the visual aggregation groups, defined for each pixel of each final subchart. Eventually, the result of the overall query contains for each group all visually dominant records that are required to produce a final visualization as obtainable from the raw data.

Note that this query requires the first series in the original query Q to have an $id = m_1$ and the last series to have an $id = m_1 + m - 1$, i.e., that the m series are numbered sequentially without any gaps in the sequence. For non-sequential series identifiers, a sequential id must be computed to allow for an equal distribution of the series to the matrix cells (cf. Section 5.4.1).

6.4.3. Pruning

Pruning of high-dimensional datasets is the second technique facilitated by the considered scaling functions (cf. Table 6.1). Knowing the number M of series that can be jointly rendered in a chart matrix, and given that $m > M$ series are selected by the original query, only M of m series should be acquired from the database and eventually displayed to the user.

Selecting a subset of series before conducting the actual data reduction not only effectively reduces the size of the query result but can also speed up the

Listing 6.1: VDDA scatter matrix query with inline parameters subquery.

```
WITH Q AS (SELECT id,t,v FROM sensors),          -- unaltered, original query
Q_p AS (SELECT $w_c w_c, $h_c h_c, $c_uv c_uv),  -- chart matrix parameters
Q_b AS (SELECT min(t)  t1, max(t)  - min(t)  dt, -- horizontal value range
              min(v)  v1, max(v)  - min(v)  dv, -- vertical value range
              min(id) m1, max(id) - min(id) dm  -- subchart assignment range
        FROM Q),
Q_g AS (SELECT id,t,v,k1 + w_c * (k2 + h_c * k3) k -- compute combined key
        FROM (SELECT *,                            -- from subkeys:
                floor( w_c  * (t  - t1) / dt) k1,  -- horizontal pixel pos.
                floor( h_c  * (v  - v1) / dv) k2,  -- vertical pixel pos.
                floor( c_uv * (id - m1) / dm) k3   -- subchart number
              FROM Q CROSS JOIN Q_p) keys),
A_scat AS (SELECT k,max(t) t_max FROM Q_g    -- compute visual aggregation
           GROUP BY k),                      -- for each display unit
Q_scat AS (SELECT * FROM Q_g JOIN A_scat     -- correlate agg. with orig. data
           ON A_scat.k = Q_g.k AND t_max = t -- via equi-join on keys & value
SELECT id,t,v FROM Q_scat                    -- remove auxiliary columns
```

query processing. Depending on the original query, the selection predicates that implement the desired pruning may be pushed down during query evaluation and potentially prevent the database from acquiring the records of the excluded series in the first place. In particular, this work considers the following pruning approaches.

First/Last Select the first/last M of m original series.

Random Select M random of m original series.

Metric Select the top M of m original series, according to a statistical metric computed for each series.

Matrix Select M of m original series, equally distributed over the available matrix cells.

The pruning is implemented, as shown in Listing 6.2, by combining one of the listed subqueries Q_{id} and the subquery Q_f with the subqueries of the VDDA query in Listing 6.1. A subquery Q_{id} first extracts a subset of ids from the original data, e.g., the first M ids in Q. Secondly, the filtering subquery Q_f joins Q with Q_{id}, restricting the original query result to the records with a matching series id. The subsequent boundary subquery Q_b and group key

Listing 6.2: VDDA subqueries for prefiltering.

```
Q_n   AS (SELECT id, count(*) n FROM Q GROUP BY id),      -- record count
Q_row AS (SELECT id, ROW_NUMBER() OVER() r                -- get row numbers
          FROM (SELECT distinct(id) FROM Q) Q_i),         -- for each id
Q_mm  AS (SELECT id, ROW_NUMBER() OVER(ORDER BY n DESC) n -- get row number
          FROM Q_n),                                      -- from record count
Q_id  AS (SELECT distinct(id) FROM Q   ORDER BY id ASC   LIMIT $M),  -- First
Q_id  AS (SELECT distinct(id) FROM Q   ORDER BY id DESC  LIMIT $M),  -- Last
Q_id  AS (SELECT distinct(id) FROM Q   ORDER BY random() LIMIT $M),  -- Random
Q_id  AS (SELECT *            FROM Q_n ORDER BY n  DESC  LIMIT $M),  -- Metric
Q_id  AS (SELECT id FROM Q_row                                      -- Matrix
          WHERE mod(r,(SELECT floor(count(*)/$M) FROM Q_row)) = 0)
Q_f   AS (SELECT Q.* from Q JOIN Q_id ON Q.id = Q_id.id)  -- filter by id
```

subquery Q_g of the VDDA query are modified to no longer include all records from the original query Q, but instead use the filtered records from Q_f.

Listing 6.2 shows the implementations for all considered pruning techniques as subqueries Q_{id}. Thereby, the *First* and *Last* subqueries extract distinct filter keys, sort them in ascending or descending order, and eventually limit the result by M. The *Random* subquery orders the filter keys by a random value, before limiting by M. The *Metric* subquery first computes a metric, such as the size of each series (using subquery Q_n), before ordering the computed filter key records by this metric and limiting by M. The *Matrix* subquery first assigns a sequence number to each series and uses a modulo comparison on the row numbers, effectively selecting one of every m/M series.

The filtering techniques can also be combined, e.g., combining the computation of a metric with a subsequent distribution to the matrix cells. For instance, the auxiliary subquery Q_{mm} in Listing 6.2 first computes the number of records per series as metric (using subquery Q_n) and assigns a row number for each record, depending on the order defined by the metric. Using Q_{mm} instead of Q_{row} in a subsequent *Matrix* subquery allows selecting M of m equally distributed series, according to the metric. This particular query ensures that series of high, medium, and low cardinality are included with equal quantities in the VDDA result set.

6.5. Scaling Function Analysis

The following section provides an analysis of the scaling behavior of the different series capacity functions, listed in Table 6.1. Therefore, the series capacity

base parameters			*computed parameters*			*lost pixel space*		
W	H	w	$h = \lfloor w \cdot q \rceil$	$u = \lfloor W/w \rfloor$	$v = \lfloor H/h \rfloor$	$W' = u \cdot w$	$H' = v \cdot h$	$loss = 1 - \frac{W' \cdot H'}{W \cdot H}$
1920	1080	48	24	40.00	45.00	1920	1080	0.00%
1920	1080	47	24	40.00	45.00	1880	1080	2.08%
1920	1080	46	23	41.00	46.00	1886	1058	3.77%
1920	1080	45	23	42.00	46.00	1890	1058	3.57%
1920	1080	44	22	43.00	49.00	1892	1078	1.64%
1920	1080	43	22	44.00	49.00	1892	1078	1.64%
1920	1080	42	21	45.00	51.00	1890	1071	2.38%
1920	1080	41	21	46.00	51.00	1886	1071	2.59%
1920	1080	40	20	48.00	54.00	1920	1080	0.00%

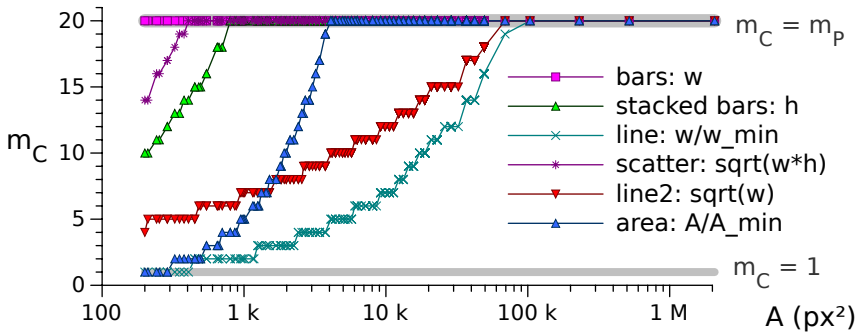
Table 6.3.: Chart matrix parametrizations and resulting lost pixel space.

functions $m_C = f(w, h)$ are visualized by plotting the value of each function over the corresponding area $A = w \cdot h$ of a subchart in the matrix. Exactly like an actual visualization, the evaluation uses discrete and constrained values for w and h, such that the resulting $u \times v$ matrix does not use more than the available $W \times H$ pixels. For some parametrizations of w and h, this leads to up to 3.8% unused pixel space, as shown Table 7.7, which contains a subset of the evaluated chart matrix configurations.

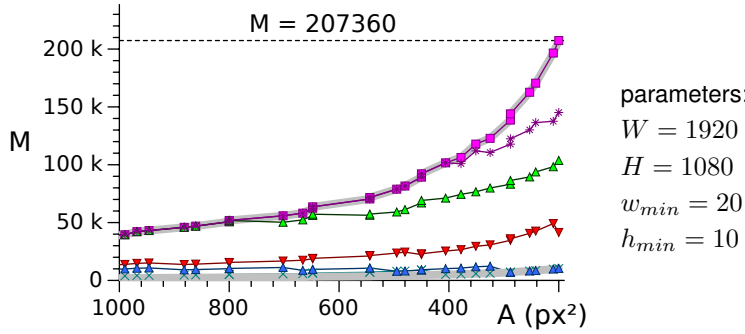
For a chart matrix with a total of 1920×1080 pixels and a minimum chart size of 20×10 pixels, Figure 6.8a shows the increasing series capacity m_C per subchart for an increasing chart size as $m_C = f(A)$. Figure 6.8b shows the corresponding matrix capacity $M = u \cdot v \cdot m_C$ as $M = f(A)$. For most chart types, m_C progresses from $m_C = 1$ to the perceptual maximum of $m_C = m_p = 20$; indicated by the gray lower and upper boundary bands in Figures 6.8a and 6.8b. The observed behavior of the different scaling functions is the following.

Scatter Plots with $m_C = min(m_P, \sqrt{w \cdot h})$ quickly scale up to $m_C = m_P$ at $A \approx 400px^2$, i.e., at $w = 29$ and $h = 14$. The scatter plot scaling function allows up to $m_C = 20$ series, even for small scatter plots, assuming that a reasonable fraction of each displayed series will be distinctly visible. Multiple series in one scatter plot often occupy only a small, potentially different portion of the pixel space, i.e., if they share the x and y axes but have different time and value distribution in the visualized range.

For scatter plots, the alternative *area* function for $m_C = min(m_P, A/A_{min})$, scales linearly with the chart area. It models a linear scaling function that



(a) Series capacity m_C over subchart area A.



(b) Matrix capacity M over subchart area A.

Figure 6.8.: Series and matrix capacity of a Full HD matrix.

always starts at $m_C = 1$ for the smallest possible chart. For the given parametrization, it scales up quickly to $m_C = m_P$.

Line Charts with $m_C = min(m_P, w/w_{min})$ slowly scale up to $m_C = m_P$ at $w = 20 \cdot w_{min}$, i.e., at $A = 400_w \cdot 200_h px^2$. In contrast to distributed point clouds in scatter plots, a line in a line chart usually spans the full width w of the chart and thus occupies at least w pixels. As a result, line charts provide the lowest series capacity.

For line charts, the alternative *line2* function for $m_C = min(m_P, \sqrt{w})$ allows for an increased amount of series also in smaller charts, while still progressing slowly to $m_C = m_P$ for a growing chart size, similar to the *line* function.

Bar Charts with $m_C = min(m_P, w)$ immediately scale up to $m_C = m_P$ at $w = w_{min} = 20px$, i.e., at $A = 20_w \cdot 10_h px^2$. Even for smaller minimal chart sizes, they would scale quickly to $m_C = m_P$, since each bar can represent another series.

Aligned Bars do not scale, since they have a constant number of series $m_C = 1$, independent of the chart size.

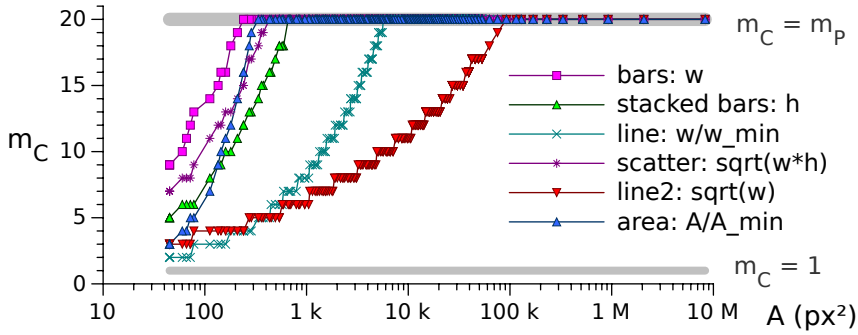
Stacked Bars with $m_C = min(m_P, h)$ quickly scale up to $m_C = m_P$ at $h = 20px$, i.e., at $A = 40_w \cdot 20_h px^2$. They are very space efficient by allowing up to h small bars to be stacked in one pixel column. The stacking reduces overplotting, e.g., compared to line charts where series are overlaying each other.

Chart Matrices

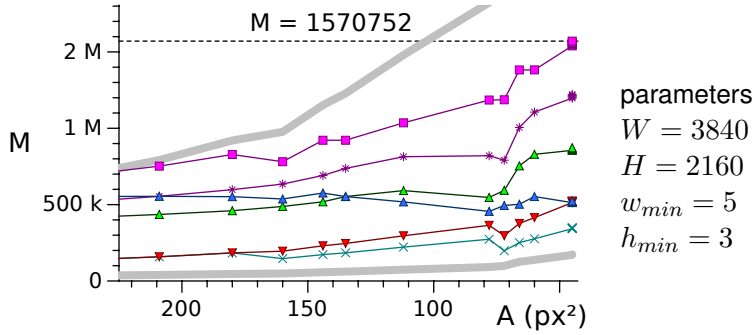
The matrix capacity M of a chart matrix is increasing for an increasing number of subcharts, even though the series capacity m_C of the shrinking subcharts is decreasing. Therefore, the chosen capacity functions in Table 6.1 are not only designed for single charts, but can be transparently used for chart matrices. In Figure 6.8a, for chart matrices with a relatively high minimum chart size of $w_{min} = 20px$ and $h_{min} = 10px$, some of the series capacity functions quickly scale up towards the perceptual series limit m_P. For comparison, Figures 6.9a and 6.9b depict the series and matrix capacity of a chart matrix of 3840×2160 pixels and a minimum chart size of 5×3 pixels. With the minimal chart size lowered to $w_{min} = 5px$ and $h_{min} = 3px$, most series capacity functions progress relatively slower towards m_P, since the chart matrix can now be composed of very small subcharts displaying significantly less than m_P series. In contrast, the *line* and *area* functions now progress faster towards m_P, since they use the now smaller w_{min} and h_{min} as divisors, effectively increasing the quotient m_C.

Monotony

All capacity functions are monotonic for most matrix configurations, i.e., for most parametrizations of W, H, w_{min}, h_{min}, and q. However, the rounding or truncation to discrete values of w_c, h_c, c_{uv}, and eventually m_C, may lead to a partially non-monotonic behavior. In particular, small values for w_{min} and h_{min} in large chart matrices may lead to non-monotonic behavior, as depicted



(a) Maximum number of series m_{max} over basic chart area A



(b) Total maximum number of series M over basic chart area A

Figure 6.9.: Scalability of UHD chart matrix.

in Figure 6.9b. The effect is caused by the increasing discretization errors with decreasing size of the subcharts.

Alternative Solution

The matrix scaling function (cf. Equation 6.3) ignores the described discretization errors and may result in non-optimal results for small w. A mitigation to this problem is to iteratively evaluate all discrete parametrizations of $w \leq W$ and $h \leq H$, and find the matrix configuration with the biggest cell size that allows displaying at least m series. Given that the aspect ratio of a subchart is fixed, only the parametrizations $w \in \{1, 2, ..., W\}$ need to be evaluated.

Summary

The developed visualization-driven data aggregation leverages overplotting as is, i.e., without considering if the resulting visualization is perceptible or not. This chapter extends this approach by moreover defining spatio-perceptual capacity functions for each chart type. Using these functions and knowing the number of acquired series, a data visualization system is able to prune the data, reducing multitudes of series to perceptible numbers that fit into the optimally configured chart matrix. Eventually, this chapter also demonstrates how the proposed capacity functions can be combined with the proposed visualization-driven data aggregation, to facilitate the pruning and subsequent visual aggregation in one integrated relational query.

7. Evaluation

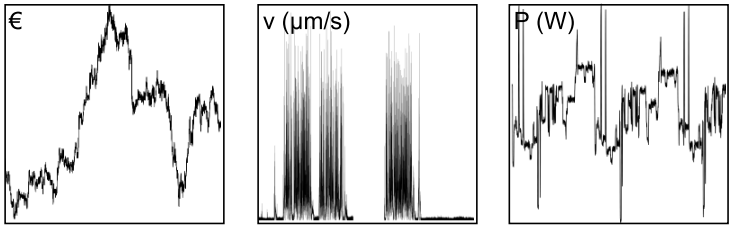
The following chapter presents an evaluation of the concepts and solutions developed in this work using prototypical implementations of the proposed query-rewriting system (cf. Chapter 3). The chapter starts by evaluating the visualization-driven data aggregation (VDDA) techniques M4 and MinMax for line charts, comparing it to several common data aggregation approaches (cf. Chapter 4). Thereafter, the chapter continues by evaluating VDDA for additional basic chart types and for chart matrices (cf. Chapter 5). After presenting an evaluation of VDDA for categorical and temporal data, the chapter concludes with an evaluation of the proposed automatic matrix configuration and pruning techniques (cf. Chapter 6).

7.1. Evaluation of M4 for Line Charts

The following evaluation compares the data reduction efficiency of the M4 and MinMax aggregation with state-of-the-art line simplification approaches and with commonly used naive approaches, such as averaging and sampling. The considered data reduction techniques are used on several real-world datasets, using the resulting visualization quality and query answer times as utility measures. All aggregation-based data reduction operators, which can be expressed using the relational algebra, are moreover evaluated regarding their query execution performance.

7.1.1. Real-World Time Series Data

This evaluation considers three different datasets: the price of a single share on the Frankfurt stock exchange over 6 weeks (700k tuples), 71 minutes from a speed sensor of a soccer ball ([81], ball number 8, 7M records), and one week of sensor data from an electrical power sensor of a semiconductor manufacturing machine ([60], sensor MF03, 55M records). Figure 7.1 depicts excerpts of the considered data.
The datasets are recorded at different data rates and have differences in their time and value distribution. The financial dataset (a) contains over 20000



(a) share price (b) ball speed (c) electrical power

Figure 7.1.: Visualization of (a) financial, (b) soccer, and (c) machine data.

tuples per working day, with the share price changing only slowly over time. In contrast to that the soccer dataset (b) contains over 1500 readings per second and is best described as a sequence of bursts. Finally, the machine sensor data (c) at $100Hz$ is a mixed signal that has time spans of low and high amplitudes.

7.1.2. Query Execution Performance

Section 4.2 described how to express simple sampling or aggregation-based data reduction operators using the relational algebra, including the proposed M4 aggregation. The different operators are now evaluated regarding their query processing performance. All evaluated queries were issued as SQL queries via ODBC over a $100Mbit$ wide area network to a virtualized, shared SAP HANA v1.00.70 instance, running in an SAP datacenter at a remote location. The considered queries are the following.

base	A baseline query selecting all tuples to be visualized.
PAA	A PAA-query computing up to $4 \cdot w$ average tuples.
round	A two-dimensional rounding query computing[1] up to $w \cdot h$ tuples.
random	A stratified random sampling query selecting $4 \cdot w$ random tuples.
first	A systematic sampling query selecting $4 \cdot w$ first tuples.

[1] The considered rounding is an implicit data reduction. The rounding query first uses f_x and f_y (cf. Equation 2.1) to project the raw data to $\mathbb{R}_w \times \mathbb{R}_h$. The result is rounded to $\mathbb{N}_w \times \mathbb{N}_h$ and then projected back to the original time and value ranges, i.e., using the inverted geometric projection functions $t = f_x^{-1}(x) = dt \cdot x/w + t_1$ and $v = f_y^{-1}(y) = dv \cdot y/h + v_1$. The actual data reduction is achieved by removing the resulting discrete duplicate values using a `distinct` selection.

MinMax A MinMax query selecting the two *bottom* and *top* tuples from $2 \cdot w$ groups.

M4 An M4 query selecting all four extrema from w groups.

Note that the group numbers are adjusted to ensure a fair comparison at similar data reduction rates. All reduction queries are modeled to produce approximately $4 \cdot w$ tuples. However, due to a small percentage of duplicated timestamps and values, some queries may select more than $4 \cdot w$ tuples. Also note that the rounding query produces significantly larger query results, since it requires the data to be grouped by pixels, instead of pixel columns, i.e., by two data columns. All queries are parametrized using a width $w = 1000$, and (if required) a height $h = 200$.

Figures 7.2a-h plot the corresponding *query execution times* and *total times* for the three considered datasets. The *total time* measures the time from issuing the query to receiving all results at the SQL client.

Each of the considered queries was issued 20 times to each of the three datasets and with varying selection predicates in the original query, e.g., selecting $70k$, $700k$, $1.4M$, and $3.6M$ baseline records. Figures 7.2a and 7.2b show exemplary results for the low volume queries. Regarding query execution time (cf. Figure 7.2a), the fastest query is the baseline query, as it is a simple selection without additional operators. The other queries are slower, as they have to compute the additional data reduction. The slowest query is the rounding query, since it requires to group the data by two data columns. The other data reduction queries only require one horizontal grouping. Comparing these execution times with the total times in Figure 7.2b, the baseline query loses its edge in query execution, ending up one order of magnitude slower than the other queries. Even for the low number of $70k$ records, the baseline query is dominated by the additional data transport time. Regarding the resulting total times, all data reduction queries are on the same level and manage to stay below one second. Note that the M4 aggregation does not have significantly higher query execution times and total times than the other data reduction queries. The observations are similar when selecting $700k$ records (30 days) from the financial dataset (Figures 7.2c and 7.2d). The aggregation-based queries, including M4, are overall one order of magnitude faster than the baseline at a negligible increase of query execution time.

The measurements show very similar results when running the different types of data reduction queries on the soccer and machine datasets. Figures 7.2e-h depict exemplary results for the high-volume tests, i.e., using $1400k$ baseline

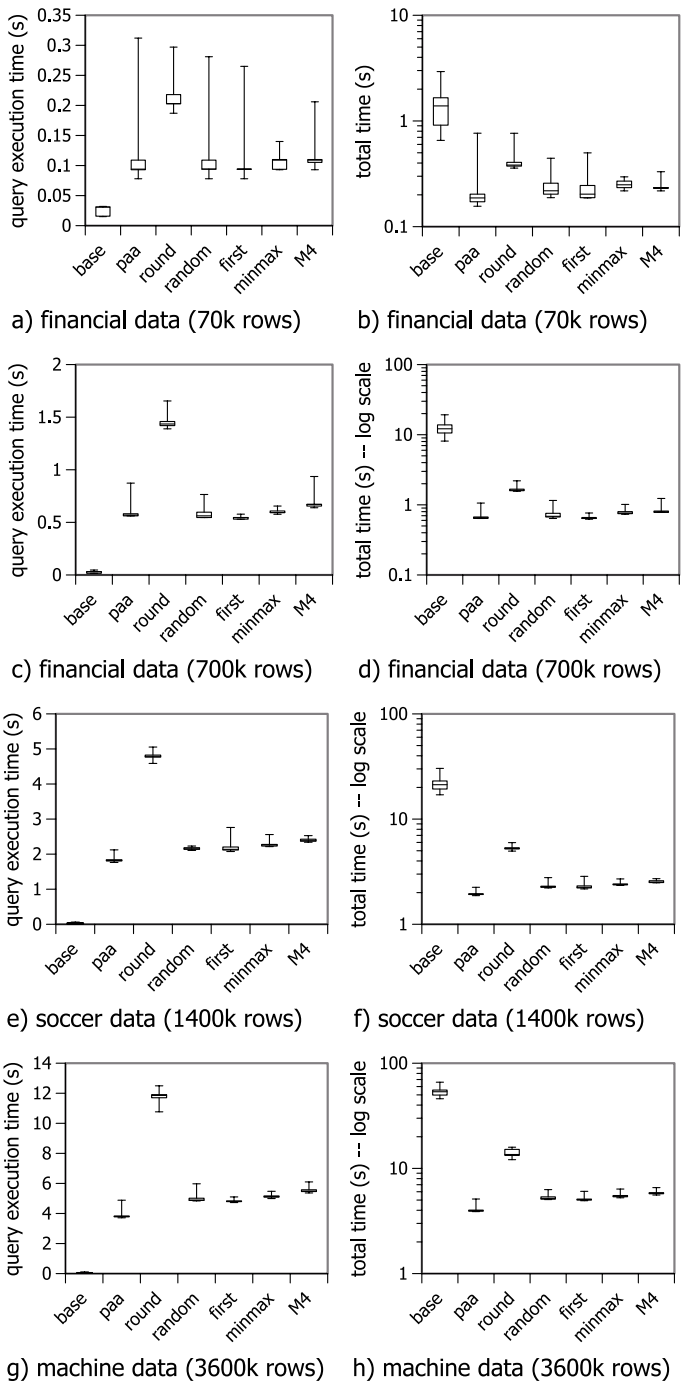


Figure 7.2.: Measured query execution times and total data transfer times.

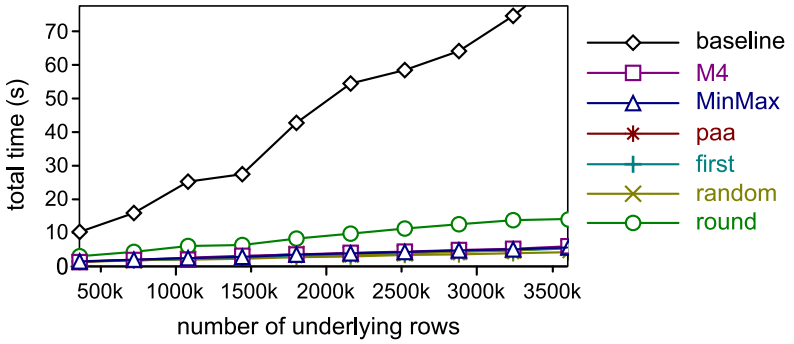


Figure 7.3.: Performance on PostgreSQL 8.4 with varying record count.

records from the soccer dataset and $3600k$ baseline records from the machine dataset. The results again show an improvement of total time by up to one order of magnitude.

All tests were repeated, using the open-source PostgreSQL 8.4.11 RDBMS running on a Xeon E5-2620 with 2.00GHz, 64GB RAM, 1TB HDD (no SSD) on Red Hat Linux 6.3, hosted in the same datacenter as the HANA instances. All data was served from a RAM disk. The PostgreSQL working memory was set to 8GB. Figure 7.3 depicts exemplary results for the soccer dataset, plotting the total time for an increasing number of underlying records. The baseline query again heavily depends on the limited network bandwidth. The aggregation-based approaches again perform much better. Queries on the finance dataset and the machine dataset provided comparable results.

The results of both the PostgreSQL and the HANA system show that the baseline query, fetching all records, mainly depends on the database-outgoing network bandwidth. In contrast, the size of the result sets of all aggregation-based queries for any amount of underlying data is more or less constant and often below $4 \cdot w$. Their total time mainly depends on the database-internal query execution performance. The evaluation also shows that the M4 aggregation is equally fast as common aggregation-based data reduction techniques. In all shown scenarios, M4 can reduce the time the user has to wait for the data by one order of magnitude and still provide the correct tuples to facilitate the rendering of a line chart, as producible from the raw data.

scenario	*data rate*	*time range*	$\|Q\|$	$\|M4(Q)\|$	*ratio*
soccer	1500Hz	60s	90k	3200	1:28
		45min (one half)	4.05M	3200	1:1266
		90min (full game)	8.1M	3200	1:2531
machine	100Hz	1h	360k	3200	1:113
		12h	4.32M	3200	1:1350
		24h	8.64M	3200	1:2700
finance	40/min	10h (business day)	24k	3200	1:8
		21d (month)	504k	3200	1:158
		252d (year)	6.05M	3200	1:1890

Table 7.1.: Data reduction rates for M4 queries on single series.

7.1.3. Data Reduction Potential

The following discussion revisits the multi-series Example 1 from Section 3.2.1, where acquiring 100 series resulted in $6M$ records and eventually $144MB$ of data, to be transferred from the database. In such scenarios, a visualization-driven data aggregation can be particularly beneficial. For instance, considering a line chart with $w = 800$ pixels, a corresponding M4 query will limit the data volume to $4 \cdot w$ records per series, i.e., to an overall maximum of $4 \cdot 800_w \cdot 100_{series} = 320k$ records, comprising only 5.3% from the original $6M$ records.

Even though Example 1 considered only 10 minutes worth of measured sensor data for each of the 100 sensors, the achieved data reduction ratio is nearly 1 : 20. In many scenarios, data reduction ratios will be even higher. For instance, assuming that an engineer may also be interested in larger time ranges, e.g., the last 12 hours for each single sensor of each monitored machine, the reduction ratios will be over 1 : 1000.

For the evaluated scenarios, Table 7.1 summarizes the achievable data reduction rates of an M4 query with $w = 800px$, considering several typical time ranges that a user may want to visualize as a line chart. The ratio is independent of the number of requested series, since the M4 aggregation is supposed to be computed separately for each series. The table shows that a visualization-driven data aggregation is not only most beneficial for scenarios with very high data rates and even when viewing only very short time ranges, but also for scenarios with low data rates of $1Hz$ and less because the visualized time ranges are often much longer in these scenarios, e.g., spanning a whole year worth of data.

7.1.4. Visualization Quality and Data Efficiency

The robustness and the data efficiency of the considered data reduction techniques can be evaluated using a sequence of data reduction queries with an increasing number of horizontal groups n_w and comparing the visualization of the query results with the visualization of the original query.

In the following evaluation, the considered queries are parametrized using $n_w \in \{1, 2, ..., 2.5 \cdot w\}$, i.e., selecting at most $2.5 \cdot (4 \cdot w)$ records and thereby more than twice as much data as is actually required for an error-free two-color line visualization. The utility measure is the DSSIM between the approximating visualization of the reduced query result and the baseline visualization of the raw data (cf. Section 4.8). The measured visualizations are drawn using the open source 2D graphics library Cairo[2]. The underlying original time series of the evaluation scenario are $70k$ tuples (3 days) from the financial dataset. The considered visualization has a width of $w = 200$ and a height of $h = 50$ pixels. Note that, for this evaluation, the number of groups n_w is allowed to be different than the width w of the visualization. This will show the robustness of the evaluated techniques. Nevertheless, in a real implementation, the engineers should ensure that $n_w = k \cdot w$ with $k \in \mathbb{N}^+$ to achieve the best results (cf. Section 4.5.3).

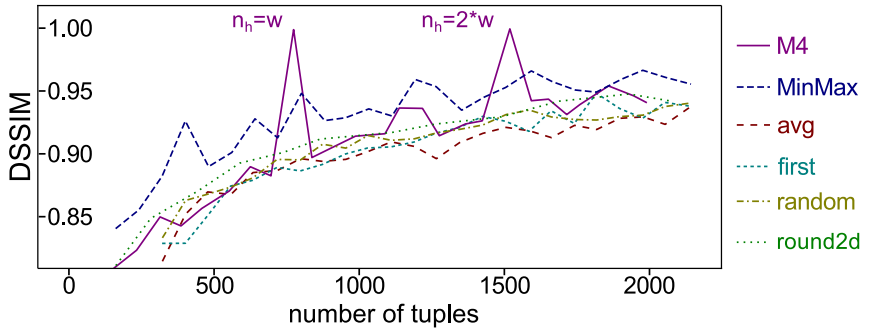
The aggregation-based operators, implementable in the relational algebra, are moreover compared to the following three procedural line simplification approaches described in Section 4.8.

ReuWi	Reumann-Witkam algorithm [89] (sequential line simplification).
RDP	Ramer-Douglas-Peucker algorithm [30, 54] (top-down line simplification).
VisWy	Visvalingam-Whyatt algorithm [100] (bottom-up line simplification).

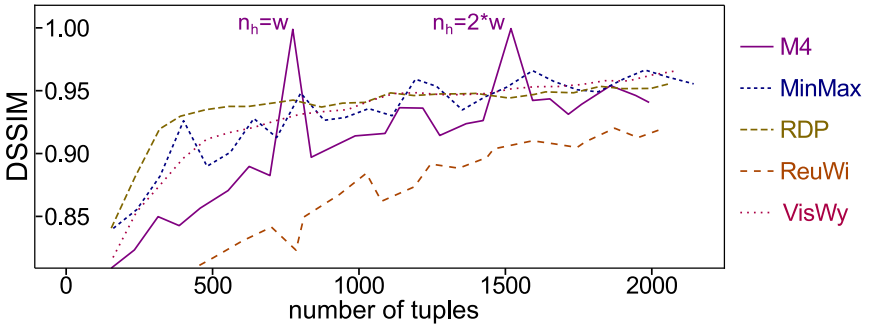
Note that RDP does not allow setting a desired data reduction ratio, so that a minimal ϵ must be precomputed to produce a number of tuples that corresponds to each specific n_w. Also note that ReuWi moreover acts as a representative for similarly sequential approaches, including APCA, which was already discussed in Section 2.7.1 and formally compared to M4 in Section 4.8.

Figure 7.4 plots the measured visualization quality (DSSIM) over the resulting number of tuples of each different grouping $n_w = 1$ to $n_w = 2.5 \cdot w$ for each

[2]Images were rendered via node-canvas (www.npmjs.com/package/canvas), using libcairo2 v1.13.0 (cairographics.org).



(a) M4 and MinMax vs. Aggregation

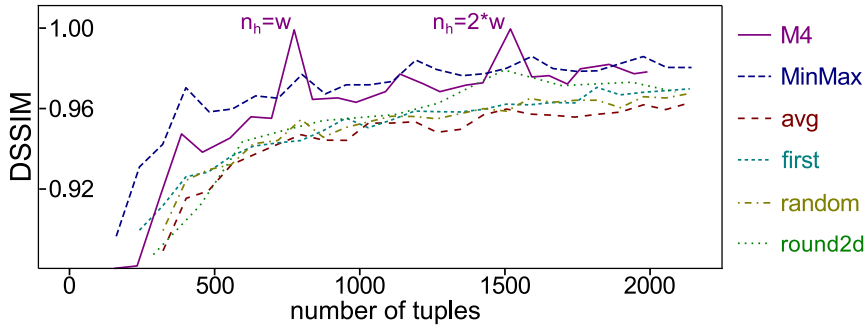


(b) M4 and MinMax vs. Line Simplification

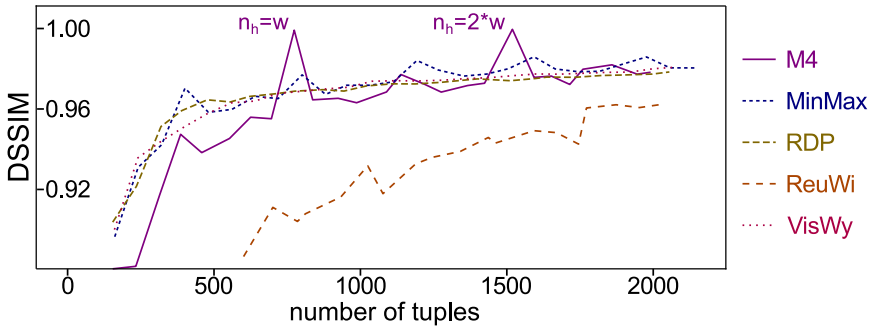
Figure 7.4.: DSSIM for binary line charts and varying reduction size.

considered data reduction technique. For readability, the plots do not show low-quality results with DSSIM < 0.8. The lower the number of tuples and the higher the DSSIM, the more data efficient is the corresponding data reduction technique for the purpose of line chart rendering. The Figures 7.4a and 7.4b depict these measures for binary line charts and the Figures 7.5a and 7.5b for anti-aliased line charts. The observed results are the following.

Sampling and Averaging operators (*avg*, *first*, and *random*) select a single aggregated value per (horizontal) group. They all show similar results and provide the lowest DSSIM. As described in Section 4.5 and moreover formalized in Section 4.5.2, they will often fail to select the tuples that are important for line rasterization, i.e., the *top*, *bottom*, *first*, and *last* tuples that are required to set the correct inner-column and inter-column pixels.



(a) M4 and MinMax vs. Aggregation



(b) M4 and MinMax vs. Line Simplification

Figure 7.5.: DSSIM for anti-aliased line charts and varying reduction size.

2D-Rounding requires an additional vertical grouping into n_h groups, with $n_h = w/h \cdot n_w$ to a have proportional vertical grouping. The average visualization quality of 2D-rounding is higher than that of averaging and sampling at comparable data volumes.

MinMax queries select $min(v)$ and $max(v)$ and the corresponding timestamps per group. They provide very high DSSIM values already at low data volumes. On average, they have a higher data efficiency than all aggregation based techniques, including M4 (cf. Figure 7.4a), but are partially surpassed by line simplification approaches (cf. Figure 7.4b).

Line Simplification techniques (RDP and VisWy) on average provide better results than the aggregation-based techniques (compare Figures 7.4a and 7.4b).

As seen previously [91], top-down (RDP) and bottom-up (VisWy) algorithms perform much better than the sequential ones (ReuWi). However, in the context of rasterized line visualizations, they are surpassed by M4 and also MinMax at $n_w = w$ and $n_w = 2 \cdot w$. Line simplification techniques often miss one of the *top*, *bottom*, *first*, or *last* tuples, because these tuples must not necessarily comply to the geometric distance measures used for line simplification (cf. Section 4.8).

M4 queries select $min(v)$, $max(v)$, $min(t)$, $max(t)$ and the corresponding timestamps and values per group. On average M4 provides a visualization quality of DSSIM > 0.9 but is usually below MinMax and the line simplification techniques. However, at $n_w = w$, i.e., at any $n_w = k \cdot w$, M4 provides error-free visualizations without pixel errors, since any grouping with $n_w = k \cdot w$ and $k \in \mathbb{N}^+$ also includes the *top*, *bottom*, *first*, and *last* tuples for $n_w = w$ (cf. Section 4.5.3).

7.1.5. The Anti-Aliasing Effect

The observed results for binary line charts (Figure 7.4) and anti-aliased line charts (Figure 7.5) are very similar. The absolute DSSIM values for anti-aliased visualizations are even better than for binary ones. This is caused by single pixel errors in a binary visualization implying a full color swap from one extremum to the other, e.g., from black (0) to white (255). Pixel errors in anti-aliased visualization are less distinct, especially in overplotted areas. In an anti-aliased line chart, a *missing line* increases the brightness of overplotted pixels, while an additional *false line* decreases the brightness of overplotted pixels. However, since lines in line charts of high-volume and thus visually dense data are heavily overplotted, the increase or decrease in brightness is often very small. Anti-aliased lines moreover rely on a partial darkening of neighboring pixels, which further reduces measurable differences caused by missing lines, false lines, and similarly small changes in overplotted areas.

To demonstrate these effects, binary and anti-aliased line renderings are compared in Figure 7.6, depicting four visualization variants of a line with five segments and illustrating how these line segments are rendered and assigned a gray value in the pixel space. Thereby, Figure 7.6a conceptually shows the four rendering variants of the considered line, shifting its geometry in sub-pixel space by 0.0, 0.25, 0.5, or 0.75 pixels to the right. Figure 7.6d conceptually illustrates how multiple neighboring lines are overplotting each other in the 2D space of the visualization. In this example, there are up to three overlaid lines

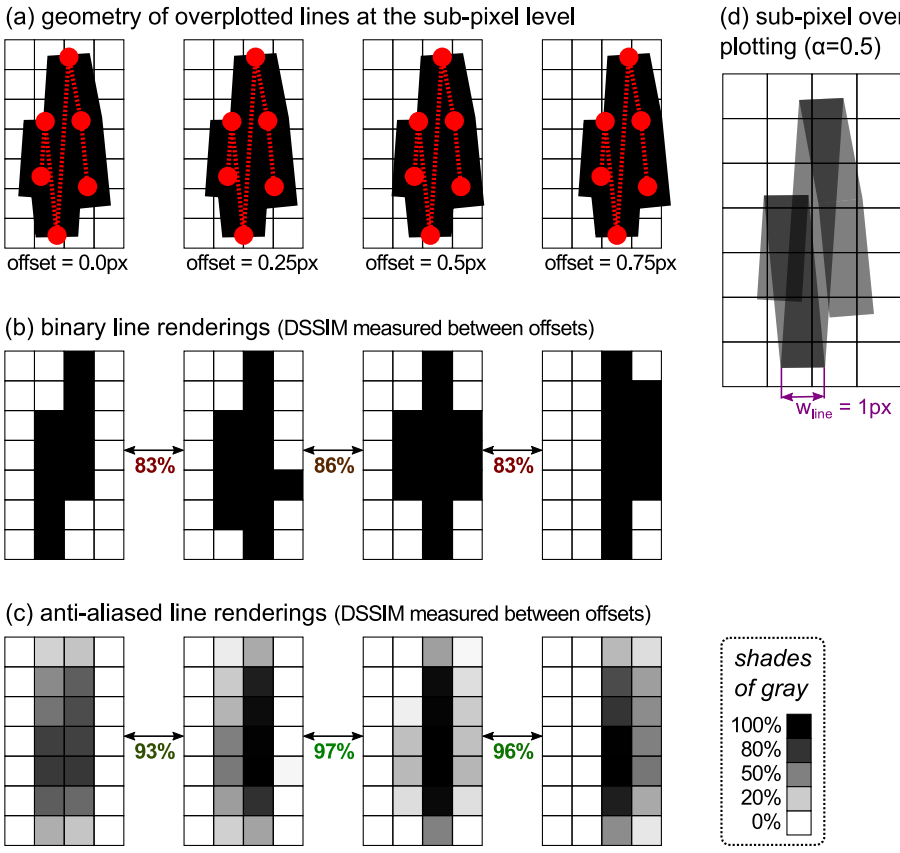


Figure 7.6.: Overplotting in anti-aliased line charts.

segments, partially overlaying each other, e.g, in the center of the second pixel column.

Based on the four given line variants, Figure 7.6b depicts the resulting pixels of the corresponding binary images. Figure 7.6c depicts the resulting pixels with their gray value, determined by the anti-aliased line rendering process[3]. Figures 7.6b and 7.6c also show the computed DSSIM between the images of each two consecutive variants. The observation is the following.

The binary images are very crisp, while the anti-aliased images are blurry. All images are trying to display the same 2D geometry and thus should be similar to each other. However, the anti-aliased renderings are more similar to each other than the binary images, as indicated by the DSSIM values in Figures

[3]The utilized renderer is the Cairo graphics library, libcairo2 v1.13.0 (cairographics.org).

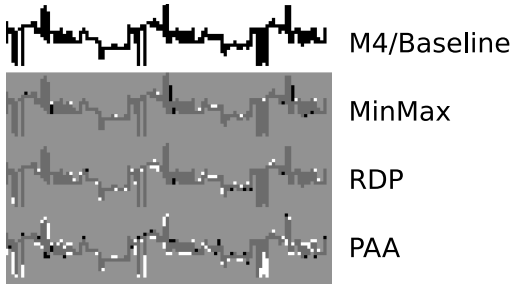


Figure 7.7.: Rendering of reduced datasets to 100x20 pixels.

7.6b and 7.6c. This shows that anti-aliased line renderings are less susceptible to small changes in the geometry of the line segments. On the contrary, binary line renderings are subject to very notable differences in the pixel space, even for small changes of the line geometry.

Essentially, the blurring of anti-aliased lines to neighboring pixels mitigates pixel errors in neighboring pixel columns and thus of missing or false lines. The blurring is moreover present and thus mitigates errors on the top and at the bottom of a line, which is the area where the most pixel errors are present, as described in the following section.

7.1.6. Evaluation of Pixel Errors

Binary line renderings of reduced datasets for the data reduction techniques M4, MinMax, RDP, and averaging (PAA) are depicted in Figure 7.7. The underlying data are 400 seconds (40k tuples) of the machine dataset. The query results are projected to small conceptual canvases of 100×20 pixels to reveal the pixel errors of each data reduction technique. M4 thereby also presents the error-free baseline image (cf. Section 4.5.2). The pixel errors are marked for MinMax, RDP, and PAA. Black pixels are additional pixels and white pixels are missing pixels, compared to the baseline image.
One can now observe how MinMax draws very long, false connection lines right of each of the three main positive spikes of the chart. MinMax also has several smaller errors, caused by the same effect. In this regard, RDP is better, as the geometric distance (cf. Section 4.8) of the not selected points to a long false connection line is also very high, and thus RDP had to split this line again. RDP also applies a slight averaging in areas where the time series has a low variance. The low variance causes the measured geometric

distances to be similarly small, resulting in many points to be omitted by the line simplification algorithm. With RDP being unaware of the pixel space, the omission often includes one of the important four extrema of a pixel column. The most pixel errors are produced by the PAA-based data reduction, caused by the averaging of the important vertical extrema. Overall, MinMax results in 30 false pixels, RDP in 39 false pixels, and PAA in over 100 false pixels. M4 provides an error-free result and thus is equal to the baseline image.

7.2. Evaluation of VDDA

The previous section already demonstrated the applicability of M4, i.e., VDDA for line charts. The following evaluation now examines the performance and discusses the visualization quality of VDDA in general.

The evaluation is based on specific original queries Q on the Frankfurt stock exchange data (cf. Figure 7.1a), with each query producing a raw data subset for a specific visualization of the data. The underlying dataset contains the 58 largest stock market shares, i.e., having the largest number of records within the evaluated day. The data has three data columns, the id of a share, the time t of a record, and the price v at that time. The complete dataset, including all 58 subsets, comprises $450k$ records and is stored in a PostgreSQL 9.3 database. The database runs on an Ubuntu/Linux 14.4 LTS installed on a 2.5Ghz Core2Duo, with 4GB RAM and an SSD. PostgreSQL uses up to 2GB working memory. The queries are issued via ODBC from the visualization client to the database. All visualizations are rendered to an HTML canvas element using the standardized drawing API of the browser (Firefox 34).

7.2.1. Evaluation Scenarios

The evaluation includes four different visualization scenarios. Each scenario is tested in multiple runs with an increasing number of underlying records, starting at $25k$ records (30 minutes), and ending at $450k$ records (9 hours).

Depending on the type of the visualization, each of the four scenarios uses a different original query, as detailed in Tables 7.2 and 7.3. In Table 7.2, the table column Q defines the original query on the underlying data $T(id, t, v)$. The column $Q \rightarrow V$ denotes how each visualization maps the result of the original query Q to the layout dimensions of the visualization V. Corresponding table columns in Table 7.3 define the relevant subqueries, required to derive a complete VDDA query, as described in Section 5.3. The considered scenarios are the following.

#	*chart type*	*original query* Q	*visual projection* $Q \to V$
1	aligned bars	$\sigma_{id \leq 12}(T)$	$(id, t, v) \to (k_u, k_{cell}, h)$
2	scatter plot	$\sigma_{id \leq 6}(T)$	$(id, t, v) \to (k_{color}, x, y)$
3	stacked bars	$_{id,t}G_{id,t \leftarrow \lfloor \frac{t}{60000} \rfloor, v \leftarrow max(v)}(\sigma_{11 \leq id \leq 15}(T))$	$(id, t, v) \to (k_u, k_{cell}, h)$
4	scatter matrix	$\pi_{id,m \leftarrow id\%3,t,v}(T)$	$(id, m, t, v) \to (k_u, k_v, x, y)$

Table 7.2.: Chart type, original query, and vis. projection for each scenario.

#	*group key subquery* Q_g	*correlated maximum aggregation*
1	$\pi_{k \leftarrow f_g(id,t),id,t,v}(Q)$	$Q_{max}(v; id, t, v)$
2	$\pi_{k \leftarrow f_g(t,v),id,t,v}(Q)$	$Q_{max}(t; id, t, v)$
3	$\pi_{k \leftarrow f_g(t),id,t,v,s}(Q \bowtie G_{t,s \leftarrow sum(v)}(Q))$	$Q_{max}(s; id, t, v)$
4	$\pi_{k \leftarrow f_g(t,v,id,m),id,m,t,v}(Q)$	$Q_{max}(t; id, m, t, v)$

Table 7.3.: Aggregation query and group key query for each scenario.

Scenario 1. A visualization of multiple data subsets as *aligned bar charts* acts as a representative for *1D charts* and *chart matrices*.

Scenario 2. A visualization of multiple data subsets in one *scatter plot* acts as a representative for *2D plots*. The scatter plot uses single pixels as marks. This scenario also represents *2D grids*, whose VDDA queries are the same as those of low-resolution scatter plots.

Scenario 3. A visualization of multiple correlated subsets in a *stacked bars chart* represents the *stacked marks* category. The required original query pre-aggregates the data from milliseconds to minutes to facilitate the stackability of the input data.

Scenario 4. A visualization of all 58 data subsets as *scatter matrix* acts as a representative for high-dimensional *chart matrices*. The considered scatter matrix uses 3×10 grid cells, resulting in multiple data subsets sharing one cell.

For each scenario, Table 7.4 lists the visualization parameters, i.e., the size $w \times h$ of the visualization canvas, the size $u \times v$ of the chart matrix, the mark type, the number of selected data subsets, and the number of different categories in the selected data.

scenario	w	h	u	v	*mark type*	*no. datasets (categories)*
1	320	20	10	1	bars	10 (1)
2	320	100	1	1	pixels	4 (1)
3	120	100	1	1	stacked bars	5 (1)
4	320	120	10	3	pixels	58 (3)

Table 7.4.: Visualization parameters and data properties for each scenario.

7.2.2. Evaluation Results

The presented results include measurements of the query execution time, i.e., the time until the query result is available to be sent from the back-end system to the local web client. The evaluation of the query execution time is complemented with measurements of the data transfer time and the rendering time at the visualization client. The sum of all three times is the total time the user has to wait until he or she can see the chart on the screen. For each scenario, and each run, the performance of a VDDA query Q_r is compared with the performance of the unreduced baseline query Q.

The observed performance of the measured VDDA queries is similar to the results obtained for the M4, the MinMax, and the other competing data aggregation queries for line charts (cf. Section 7.1). The measurement results are depicted in Figure 7.8, showing the query execution times, data transfer times, and rendering times for each test run.

The results show that the execution time of all VDDA queries is not significantly slower and often even notably faster than the execution time of the baselines queries, even though the VDDA queries require the database to compute the additional data aggregation. This shows that VDDA provides a measurable benefit even before the data is transferred from the database. Regarding the total query answer time, until the data is visible at the visualization client, the VDDA queries of Scenarios 1, 2, and 4 are even up to three times faster than the baseline. This significant speedup is obtained from notable improvements of all three measured times, the query execution time, the data transfer time, and the rendering time.

Depending on the original query, the notable speedup of query execution times is facilitated by the database having to acquire and serialize less data to be sent to the downstream components. The significantly faster data transfer times are obtained, since less data has to be transferred over the network and deserialized by the visualization client. The overall improvement is com-

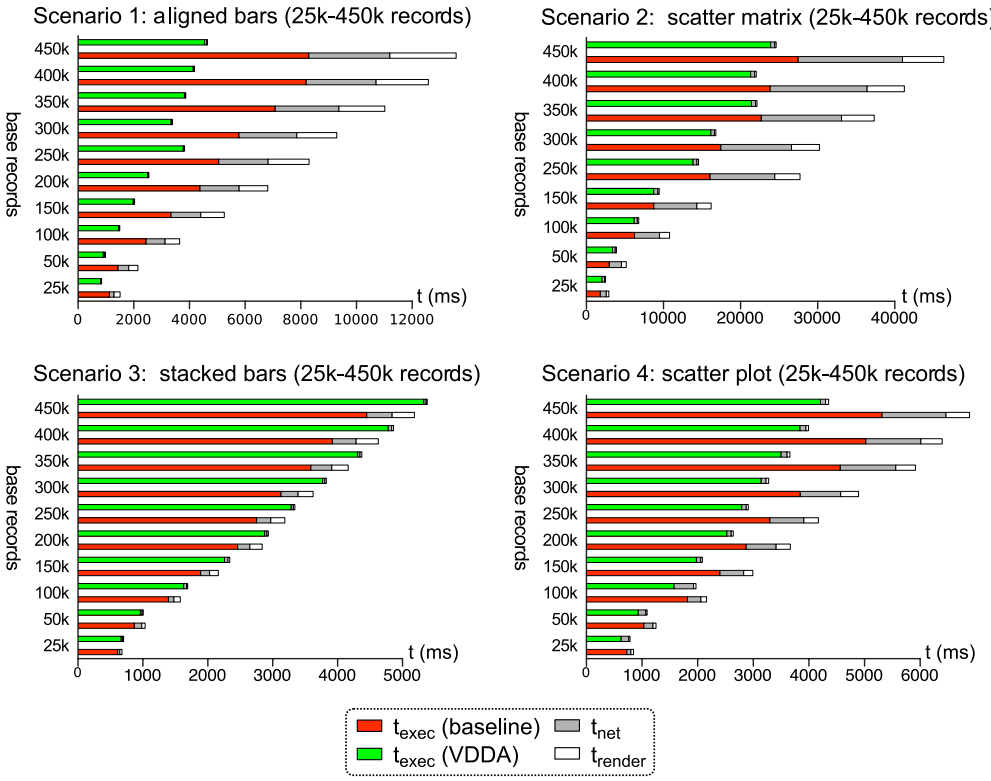


Figure 7.8.: Performance of VDDA and original queries.

plemented with significantly faster rendering times, since the reduced data volumes require fewer geometric primitives to be drawn on the screen.

The measured improvements are relatively higher for queries on high numbers of underlying records. However, even for small datasets of $25k$ underlying records, the VDDA queries are faster and already provide a small benefit. The higher the number of underlying records, the higher is the speedup of VDDA over the baseline.

The achieved speedup and data reduction ratios are moreover summarized in Table 7.5, listing the results for the first run with $25k$ underlying records and the last run with $450k$ underlying records for each scenario. The overall average performance increase of all measured VDDA queries over all runs for the different scenarios is 60%, i.e., the data reduction queries are answered 1.6 times faster than the original queries. The average data reduction ratio is $1:43$, with $1:1.4$ at the lower end and $1:254$ at the higher end.

scenario	*speedup* (25k records)	*reduction* (25k records)	*speedup* (450k records)	*reduction* (450k records)
aligned bars	1.8	1:16	2.9	1:254
scatter plot	1.2	1:3	1.9	1:33
stacked bars	1.0	1:1.4	1.0	1:20
scatter matrix	1.1	1:1.7	1.6	1:13

Table 7.5.: Speedup and data reduction of VDDA over the baseline.

7.2.3. Visualization Results

All VDDA operators, defined for the considered chart types, use a correlated maximum aggregation (CMA) to select the dominating records per display unit (cf. Section 5.3.5). For most chart types this maximum aggregation effectively selects all records required to produce all correct pixels, so that the visualization of the reduced data is exactly the same as the visualization of the original data. This is also the case for the four evaluation scenarios.

Scenario 1. For bar charts, only one maximum record per pixel column is required to draw the correct bar for each column. The visualizations of three runs are depicted in Figure 7.9, which also lists the size of the base relation $|T|$, the size of the original query result $|Q|$, and the size of the VDDA query result $|Q_r|$ for each of the three runs. Note that the VDDA query usually selects more than $w = 320$ records, since the correlated aggregation does not remove duplicates (cf. Section 4.7.1).

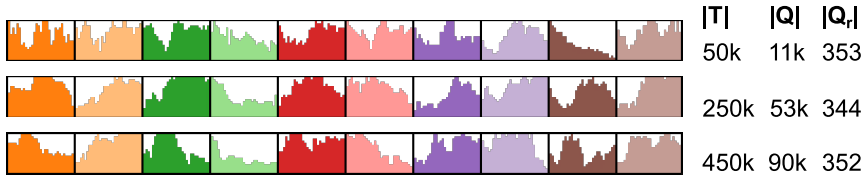


Figure 7.9.: Scenario 1: aligned bars, ten series in separate charts.

Scenario 2. For scatter plots with single pixel marks, only one *last* (maximum time) record per pixel is required to draw a correct pixel, i.e., given that the rendering process consumes the records ordered by time. The visualizations and cardinalities of three runs are depicted in Figure 7.10.

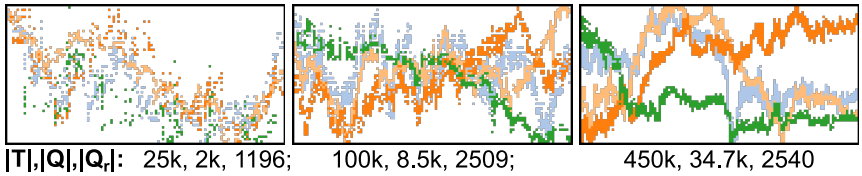


Figure 7.10.: Scenario 2: scatter plot, four series in one plot.

Scenario 3. For stacked bars charts, the records defining the highest stack of bars are required. These records are obtained from computing the highest *sum* of values of the underlying series per timestamp, and correlating the result with the base relation, to effectively select all records that contribute to that sum. The visualizations and cardinalities of three runs are depicted in Figure 7.11.

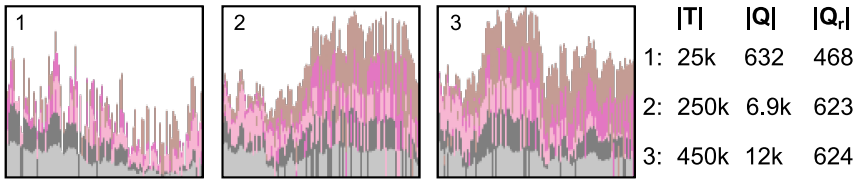


Figure 7.11.: Scenario 3: Stacked Bars, four series in one chart.

Scenario 4. A scatter matrix is a composition of scatter plots. For scatter plots with single pixel marks, the VDDA query selects the *last* record per pixel. This results in an error-free visualization, given that the rendering process again consumes the records ordered by time. The visualizations and cardinalities of two runs are depicted in Figure 7.12.

For all scenarios, one can observe that the number of records produced by the VDDA queries Q_r are virtually the same as the number of drawn pixels or the number of bars of the visualization. The additional records that would be selected by the original queries do not provide any benefit and also require the client-side renderer to redraw many pixels or bars dozens to hundreds of times.

In summary, VDDA is able to produce correct visualizations for the most common chart types, at lower query execution times, lower data transfer times, and lower rendering times, compared to the times measured for acquiring the raw underlying data using the original queries.

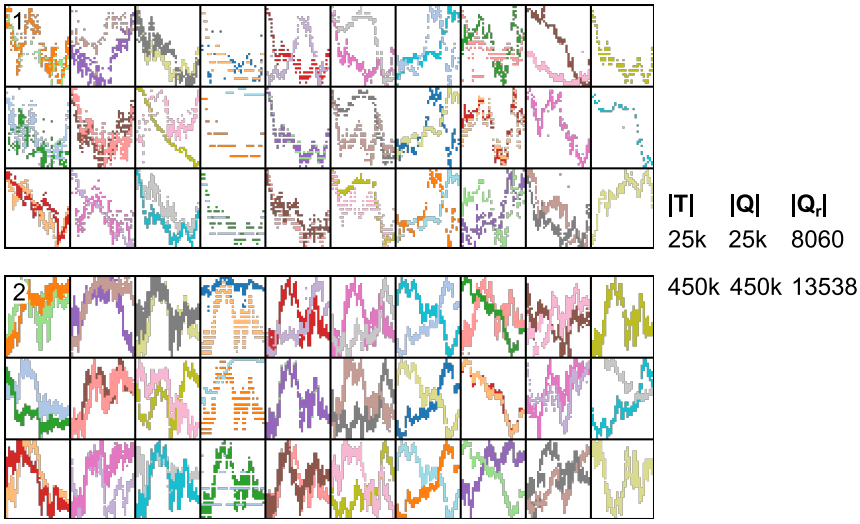


Figure 7.12.: Scenario 4: Scatter Matrix, 58 series in 3x10 matrix.

7.3. Temporal, Categorical, and Spatio-Perceptual VDDA

The following section first evaluates the applicability of the categorical and temporal extensions of VDDA, followed by an evaluation of the developed automatic chart configuration for VDDA in general. The evaluation setup is the following.

7.3.1. Evaluation Setup

All measurements were conducted on a quad-core MacBook Pro, 2.5Ghz Intel Core i7 with 16GB RAM, using the manufacturing dataset [60] (cf. Section 7.1.1). The evaluation scenarios are the same as in the previous section, considering the same chart types and slightly modified original queries. However, this evaluation now uses an increased number of series, i.e., using a baseline of 3200 subranges of the manufacturing dataset, instead of the previously used 58 share prices. The underlying data now comprises $32M$ records. To reflect this change, all queries are modified to select a higher number of series, by altering the original series selection predicates by a factor of 100, e.g., resulting in the original query $Q = \sigma_{id \leq 1200}(T)$ for the first scenario (cf. Table 7.2). To cover the most common chart types, the four previous scenarios are complemented with a fifth line chart scenario. Instead of single charts, chart matrices are

scenario	*chart type*	Q	$\lvert Q\rvert$	m	u	v	$\frac{m}{u \cdot v}$
1	bar chart	$\sigma_{id \leq 1200}(T)$	12M	1200	20	60	1
2	scatter plot	$\sigma_{id \leq 600}(T)$	6M	600	10	10	6
3	stacked bars	$\sigma_{1100 \leq id \leq 1500}(T)$	4M	400	4	10	10
4	scatter plot	T	32M	3200	20	20	8
5	line chart	$\sigma_{id \leq 160}(T)$	1.6M	160	4	4	10

Table 7.6.: Considered chart types and corresponding queries.

used for data visualization, with manually or automatically defined values for u and v, to distribute the multitude of selected series to the matrix cells.

Table 7.6 lists, for each evaluation scenario, the *chart type*, the considered original query Q, and the number of non-aggregated records $|Q|$ produced by Q, the number of selected series m, the number of matrix columns u and rows v, and the resulting number of series per cell $m/(u \cdot v)$. For the evaluation, each original query Q in Table 7.6 is rewritten to a VDDA query, according to the type of the visualization (cf. Section 5.3). Note that the given fixed values for u and v are only used for defining the baseline VDDA queries in the evaluation of the temporal and categorical conversion. The subsequent evaluation of the chart matrix configuration and pruning techniques will use variable values for u and v.

7.3.2. Temporal and Categorical Conversion

Section 5.4.3 described how to handle SQL dates and timestamps in VDDA queries. The following section now evaluates the overhead of the required temporal arithmetic, by using two variants of the same datasets, based on a time series $T_t(id, t, v)$, with $id \in \mathbb{N}$ and $t, v \in \mathbb{R}$, and a time series $T_{ts}(id, ts, v)$, where ts is an SQL timestamp. Section 5.4.2 moreover described how to convert categorical data to numerical data, as required for the VDDA grouping function. Therefore, the time series T_{ts} is altered to derive a third time series $T_c(c, ts, v)$, replacing the numeric id column with a string column c.

For each evaluation scenario (cf. Table 7.6), Figure 7.13 compares the query execution times of a baseline VDDA query Q_t on T_t, a temporal arithmetic VDDA query Q_{ts} on T_{ts}, and a temporal arithmetic plus categorical conversion VDDA query Q_c on T_c. The results are the following.

For all five scenarios, one can observe the baseline VDDA queries Q_t being the fastest. As expected, the temporal arithmetic queries Q_{ts} are slower. However, since the temporal arithmetic requires only one additional operation

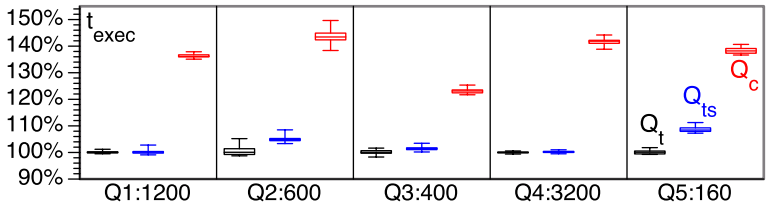


Figure 7.13.: Overhead of temporal and categorical queries over the baseline.

on each selected record of the original query Q, the overhead is only 3% on average and 11% at most. Categorical conversion is more expensive, with up to a maximum of 50% additional execution time and an average overhead of 37%. The overhead is caused by the additional join of the filtered series *id*s with the original data, required to assign a correct category number to each record.

7.3.3. Scalability-Based Pruning

This final evaluation uses the automatic chart matrix configuration and examines the benefit of using the scalability-based pruning in combination with VDDA. Therefore, a series of VDDA queries is issued for different chart matrix configurations, implementing the different pruning techniques, as described in Section 6.4.3.

The corresponding VDDA queries are defined for each scenario (cf. Table 7.6), using an increasing number of cells and thus decreasing size of the cells. For brevity, the possible parametrizations of the chart matrices are limited to the nine parameter sets (w, h, u, v, M) defined in Table 7.7, and one additional automatic parametrization $(w^*, h^*, u^*, v^*, M^*)$, computed by the developed automatic chart matrix configuration technique (cf. Section 6.4.1).

Note that all query results are technically limited to $2M$ records, even though a multi-series VDDA query without pruning may produce significantly more than $2M$ records. For comparison, the column $|M4_{all}|$ in Table 7.7 lists the underlying number of records selected by a non-pruning, unlimited M4 query for a corresponding line chart matrix, i.e., including *all* 3200 series in the query result and not considering the visual scalability of the matrix or any technical limits. This shows that trying to visualize a multitude of series in matrices with only a few cells, i.e., with very large w and h, may easily result in the whole dataset being acquired from the database by a non-pruning VDDA query. In

w	h	m_C	u	v	M	$\lvert M4_{all}\rvert$
1920	1080	20	1	1	20	16.3M
960	540	20	2	2	80	8.6M
480	216	20	4	5	400	4.5M
240	120	12	8	9	864	2.4M
120	60	6	16	18	1728	1.3M
60	30	3	32	36	3456*	733k
30	15	2	64	72	9216*	410k
20	10	1	96	108	10368*	275k
5	3	1	384	360	138240*	70k

*allows to include all 3200 series (no pruning)

Table 7.7.: Chart matrix configurations with series and matrix capacity.

the following, this worst case $M4_{all}$ is included as additional Scenario 6, with an original query $Q = T$.

Figures 7.14 and 7.15 depict the measurements of one experiment with the 9+1 corresponding VDDA queries for all six scenarios, using several exemplary pruning techniques, denoted as follows.

all No pruning.

first Select the first M of m series.

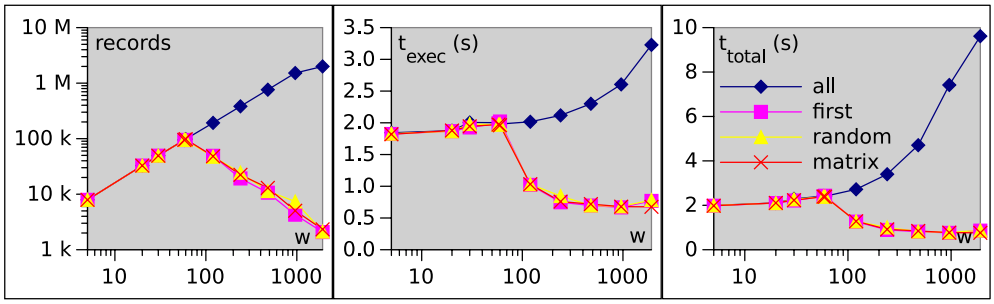
random Select M of m series randomly.

matrix Select M of m series evenly distributed, i.e., every $(\frac{m}{M})^{th}$ series.

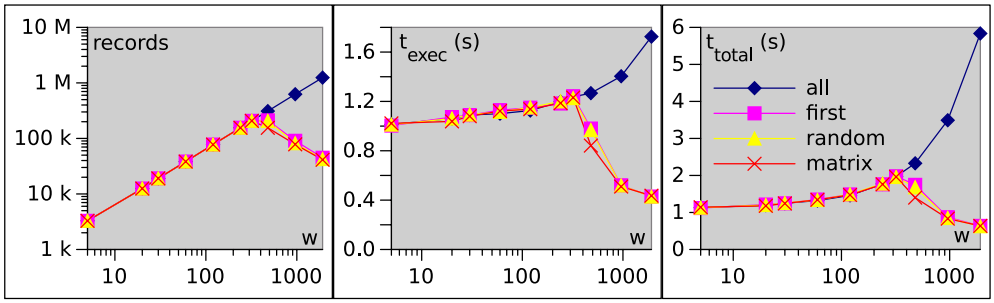
The figures show three consecutive plots for each scenario. The first shows the number of *records* of the VDDA query result, the second the query execution time t_{exec} spent in the database, and third the total query answer time t_{total} until the data is available at the visualization client. Note that t_{total} does not include the overhead of the counting query, required to obtain the actual number of series selected by the original query. For all scenarios, the overhead, including the time for the automatic chart configuration was $< 200ms$. The observation is the following.

Aggregating over all 3200 series is the most expensive operation, especially for large chart sizes, since the number of records per series increases with increasing chart size. The non-pruning baseline queries (*all*) are up to one order of magnitude slower than the pruning queries. Note that the technical limit of $2M$ records influences the total time, e.g., at $w = 1920$ in Scenario 1 and for $w \geq 240$ in Scenario 6, since fewer records have to be serialized and

Scenario 1: aligned bars, $Q = \sigma_{id \leq 1200}(T)$, $w^* = 58$



Scenario 2: scatter matrix, $Q = \sigma_{id \leq 600}(T)$, $w^* = 320$



Scenario 3: stacked bars matrix, $Q = \sigma_{1100 \leq id \leq 1500}(T)$, $w^* = 480$

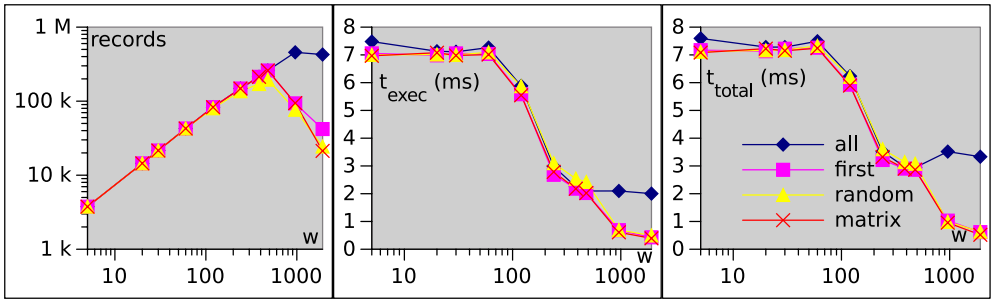
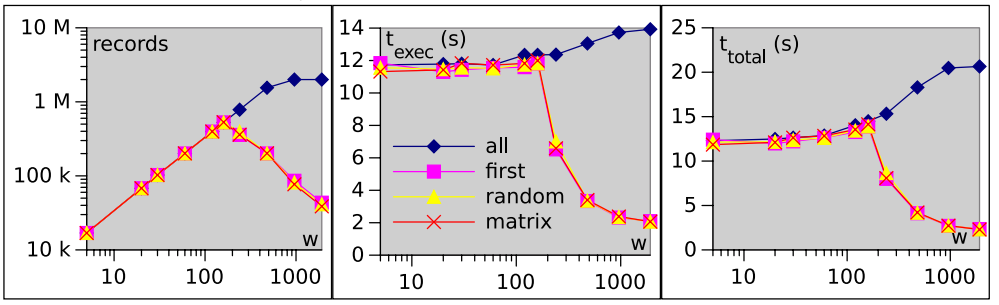
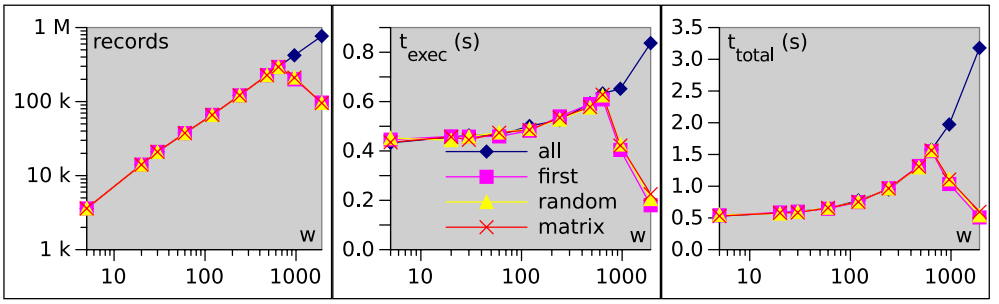


Figure 7.14.: VDDA performance and cardinality results (Scenarios 1-3).

Scenario 4: scatter matrix, $Q = T$, $w^* = 160$



Scenario 5: line chart matrix, $Q = \sigma_{id \leq 160}(T)$, $w^* = 640$



Scenario 6: line chart matrix, $Q = T$, $w^* = 71$

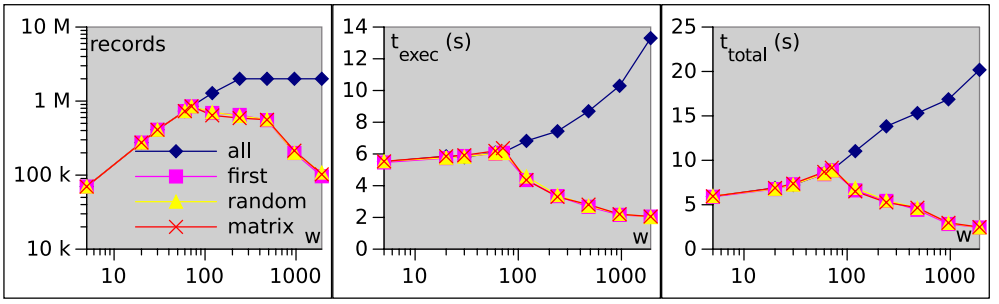


Figure 7.15.: VDDA performance and cardinality results (Scenarios 4-5).

transferred to the client. However, technically limiting the output of the data aggregation does not influence the execution time, since the data aggregation is computed on the complete result of the *preceding* original query Q, which may still comprise up to $32M$ records.

The pruning VDDA queries in all scenarios are on par with the baseline queries up to a chart width of $w \leq w^*$, i.e., they select the same number of records and exhibit similar query execution and total times. At $w \leq w^*$ the matrix capacity M of a corresponding chart matrix is higher than the number of actually selected series m, i.e., no series are pruned. However, at $w > w^*$, M quickly decreases, as the number of matrix cells decreases. Consequently, all pruning queries effectively exclude an increasing number of series from the data aggregation, resulting in a significant speedup of t_{exec} and t_{total} over the baseline of up to one order of magnitude. The data reduction of the pruning queries over the baseline is up to three orders of magnitude, e.g., for $w = 1920$ in Scenario 6. This observation confirms the results of the M4 evaluation in Section 7.1.

All scenarios show similar results, with the query execution times and total times dropping significantly below the baseline at $w > w^*$, gaining a speedup of up to one order of magnitude. This also holds for Scenario 3, even though the query execution time decreases for an increasing width w of the charts of the matrix, while it increases for all other scenarios. This unexpected behavior is caused by the additional subqueries required for stacked marks (cf. Section 5.3.2). In particular, for small w and thus for a large number $u \cdot v$ of cells, the stacking subquery requires to first compute the stack sums for all w pixel columns of all $u \cdot v$ cells and thus requires an aggregation of the data in up to $w{\cdot}u{\cdot}v = 5{\cdot}384{\cdot}360 = 691200$ buckets. This particularly large aggregation result is subsequently joined with the original query result to assign the computed sums to each record. For large w the number of aggregation buckets is much smaller, e.g., $w^* \cdot u \cdot v = 480 \cdot 4 \cdot 5 = 9600$ at the break-even point of Scenario 3. This subsequent join can be computed much faster. Note that the joins for the described correlated aggregation are usually processed as hash joins that can be computed in $\mathcal{O}(n + m)$, i.e., in $\mathcal{O}(|Q| + w \cdot u \cdot v)$ for the described scenario.

The observation generally shows that the overhead incurred by the additional pruning logic is negligible and the different pruning techniques moreover have nearly identical performance, since the pruning is computed on the small number of $m \leq 3200$ distinct series *id*s, rather than on the complete dataset. Independent of the chosen pruning technique, the result of the pruning sub-

queries Q_{id} (cf. Listing 6.2 on page 145) are always $M \leq m$ filtered series *id*s, to be joined with the original query result Q for filtering the actual records.

Summary

This chapter evaluated the applicability of M4 in particular and of VDDA in general, demonstrating a data reduction by several orders of magnitude and improving query answer times by up to two orders of magnitude. The chapter also evaluated the overhead of converting non-numeric data in SQL databases, observing a negligible overhead for timestamp conversion and a more notable overhead for converting categorical data. Eventually, this chapter also demonstrated the benefits of the proposed chart matrix configuration and pruning techniques, allowing for an additional reduction of the data by one or more orders of magnitude with corresponding improved query answer times.

8. Conclusion

This work analyzed techniques for data reduction in context of data visualization and developed methods to better leverage the knowledge about the desired visualization for driving a related data reduction process. The goals of this work (cf. Section 1.2) were to provide reduced datasets of a predictable size, which are computed inside the database, transparent to the visualization client, and incurring only a low approximation error. Thereby, the data reduction should be fast to compute and facilitate high data reduction ratios.

The following chapter now summarizes the contributions, describing how this work achieves the described goals, solves the research question, and fulfills the research hypotheses (cf. Section 1.2). The chapter concludes with an outlook on future work.

8.1. Research Contributions

Focusing on the data processing capabilities of relational database management systems (RDBMS) and considering them as essential parts of data visualization systems (DVS), this work developed the following solutions.

Visualization-Driven Transparent Query Rewriting. The proposed query rewriting approach redefines how a DVS accesses data in an attached database. Driven mainly by the parameters of the visualization, in particular its width w and height h, the described solution provides data-reduced query results of predictable size (Goal 1). Thereby, the rewriting does not alter the schema of the data and is thus transparent to the visualization client (Goal 2). The considered data reduction is defined at the query level (Goal 3), relying only on a few additional parameters and not requiring to alter the client's original query. Therefore, the presented solution is non-intrusive and can be easily integrated in any kind of modular DVS.

Visualization-Driven Data Aggregation for Line Charts (M4). The core idea of the work was to model the pixel-level rendering of a visualization as query-level

data aggregation. Focusing on line charts, as the most ubiquitous visualization of high-volume numerical data, the main contribution of this work is the in-depth analysis of the line rendering process and the resulting pixel-level properties of line charts, in the context of data aggregation. The provided analysis facilitated the development of the M4 aggregation that complements the proposed query rewriting by accomplishing the remaining goals. Thereby, M4 supersedes any existing approach by providing pixel-perfect visualizations of reduced datasets, as obtainable from the raw data (Goal 4). M4 can moreover be computed in $\mathcal{O}(n)$, and thus it is fast (Goal 5). Finally, M4 can reduce any incoming high-volume numerical dataset to a very small number or $4 \cdot w$ records (Goal 6).

Visualization-Driven Data Aggregation for Common Charts (VDDA). Based on the principles used to develop the M4 aggregation, this work defines a general model to define a visualization-driven data reduction for any kind of chart type. Therefore, this work generalizes and defines the essential components of a VDDA query and subsequently uses these to model an appropriate data reduction for the most common chart types to essentially simulate or approximate the rendering process of each chart type. Providing solutions for basic charts and chart matrices alike, the defined VDDA queries can be used by system builders to cover and improve a broad range of data visualization scenarios. The evaluation of VDDA has shown that this predictable, query-level, and transparent data reduction (Goals 1-3) involves only a low approximation error, is fast to compute, and can moreover significantly reduce large data volumes (Goals 4-6).

Automatic Chart Matrix Configuration and Capacity-Based Pruning. This work showed that incorporating additional perceptual properties is essential for visualizations of large multitudes of time series. Therefore, following the developed chart type taxonomy, this work defines specific data capacity functions for the main chart types and demonstrates their applicability for data reduction and visualization automation. With VDDA incorporating the proposed capacity-based pruning, this work introduces a novel visual aggregation approach that can be used to effectively, efficiently, and correctly visualize virtually any kind of big data.

8.2. Hypotheses Fulfillment

Answering the main research question (cf. Section 1.2), the discussion and development of the M4 aggregation in particular and the development of VDDA in general have shown how a visualization-inherent data reduction can be modeled as explicit data aggregation.

With the M4 aggregation for line charts and VDDA for the remaining common chart types defined as visualization-driven and query-level data aggregation, this work provides solutions for any kind of big data visualization, essentially fulfilling research hypothesis 1.

The discussion and evaluation of query execution performance, total data transfer times, rendering speed, and visualization quality have shown the superiority of the approach. Fulfilling research hypothesis 2, VDDA provides significantly reduced total data transfer times, at a negligible expense of query execution time, and without impairing the visualization quality for common types of data visualizations.

8.3. Future Work

With big data recorded in a broad range of scenarios, techniques for a reduction of transferred data volumes will become an essential part of all kinds of software products. However, as this work has shown, generic measures for data reduction, such as random sampling and averaging, are not desirable for big data visualization systems. Therefore, future data analysis and data visualization software must learn to better respect and analyze the detailed requirements of the application scenarios. Paying particular attention to the previously neglected endpoint of the visualization pipeline, i.e., considering the pixel-level properties of a data visualization, future work can extend and improve the present work in a variety of aspects.

Extending the VDDA Model

The developed VDDA operators in general and the M4 aggregation in particular demonstrated how to simulate or approximate the pixel-level rendering of geometric shapes as query-level data aggregation. However, the developed operators are very chart-specific, e.g., by assuming only single pixels as marks in a scatter plot or by focusing on binary line charts with a line width of only one pixel. Considering the plethora of data visualizations and rendering techniques used by thousands of software products today, the developed visual

aggregation models are only the first starting point to be extended frequently or refined in unexpected ways.

Thereby, future VDDA extensions may either further incorporate the properties of the underlying data, e.g., analyzing the actual data distribution, or they may leverage more sophisticated visualization models, e.g., considering the size and style of graphical marks. For instance, a visual aggregation for a bubble chart may analyze the records of the radius-defining data column to more precisely simulate overplotting of neighboring pixels at the query level.

Supporting Additional Chart Types

This work follows Mackinlay's chart taxonomy [77] and thus restricts the types of visualizations to a reasonable but limited set. Some user may be missing infamous pie charts, scientific radar charts, or powerful treemaps. In general, using the principles developed in this work, future research may define and evaluate the visual aggregation inherent in many additional kinds of common and uncommon data visualizations.

Finally, even though they are often less useful for data visualization, visual aggregation models may be developed for 3D visualizations, e.g., incorporating the information about an objects depth to detect if it is occluded by other objects in the 3D space of the visualization. Similar tests are already performed in hardware on modern graphics processing units and may thus be leveraged for the data reduction.

Using Complementary Data Reduction Techniques

The methods developed in this work specifically focus on the visual aggregation and are supposed to work for any amount of underlying records. However, in practice, additional means for data reduction have to be considered, e.g., even before storing and providing access to the data. In this regard and as discussed in Sections 2.6 and 2.7, VDDA may be combined with complementary data reduction techniques, e.g., to provide pre-aggregated data [75] organized in hierarchical [31] or amnesic [44] data models.

Leveraging VDDA for Interactive Visualizations

This work primarily assumes queries to be dynamic [93] and thus issued as ad-hoc queries to the DBMS. This may result in consecutive queries requesting overlapping data ranges. To avoid this shortcoming, future visualization

systems may combine VDDA with data management techniques for interactive data visualizations [94, 53], as briefly discussed in Section 2.8.4 of this work.

Studying the Perceptibility of Data Visualizations

Chapter 6 discussed additional problems of data visualizations, regarding their perceptibility, and developed the first formal solution to these problems. However, the presented automation technique, i.e., the chosen data capacity functions, are based on a simplification of the problem, i.e., they assume that different datasets behave similarly when rendered to the pixel space of a visualization. Unfortunately, the data distribution in real-world data can be very heterogeneous and the assumed multi-series overplotting behavior may not hold in general. In this regard, future work may consider other types of big data than the three sensor datasets considered and evaluated in this work (cf. Table 1.1 and Section 7.1.1). Future research on visual aggregation techniques [103, 107] and visualization-related data reduction requires a more comprehensive survey on the perceptibility of data in data visualizations. This survey should cover all kinds of big data, analyze the data-specific overplotting behavior of common data visualizations, and eventually measure their perceptual scalability. As this task may provide research results for one or more dissertations, researchers from the areas of data management, computer graphics, and human-computer interaction should join their forces on that account.

Towards a Perception-Driven Data Reduction (PDDR)

Complementary to the pixel-level properties, there are the physical properties of a data visualization that a *perception-driven data reduction* may consider, e.g., to define different smallest *display units* for a visual aggregation of the data. These display units may be smaller or larger than a pixel, not necessarily rectangular, and may also overlap each other. Eventually, these units should have a counterpart in the perception model of the human brain, which can already be leveraged to construct data visualizations of unprecedented perceptibility [84]. Consequently, future work on visual aggregation may leverage this model, allowing for a visual aggregation not only at the pixel level, but in the smallest perceivable display units of the visualization processing system of the human brain. The development of a corresponding generic and perception-aware DVS is a challenging task for future work, for which the present work may provide the first cornerstone. Essentially anticipating how the user perceives the transferred and visualized data, and consequently leveraging this

information for always sending a perfectly perceivable fraction of the data, may be considered as the most sophisticated and desirable solution for visually reducing big data in a data visualization system.

List of Figures

List of Tables

List of Algorithms

Listings

References

[10] Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation), ISO/IEC 9075-2:2003. page 303, 2003.

[11] SAP HANA SQL and System Views Reference. Technical report, May 2016. Document Version: 1.0 2016-05-11.

[12] D. J Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR '05*, pages 277–289. VLDB Endowment, 2005.

[13] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12):1902–1905, 2012.

[14] W. Aigner, S. Miksch, H. Schumann, and C. Tominski. *Visualization of Time-Oriented Data*. Human–Computer Interaction Series. Springer, 2011.

[15] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Processing high data rate streams in System S. *Journal of Parallel and Distributed Computing*, 71(2):145–156, 2011.

[16] L. Battle, M. Stonebraker, and R. Chang. Dynamic Reduction of Query Result Sets for Interactive Visualization. In *IEEE Big Data '14*, 2013.

[17] J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, 1983.

[18] E. Blais, A. Kim, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid Sampling for Visualizations with Ordering Guarantees. *PVLDB*, 8(5):521–532, 2015.

[19] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *TVCG*, 17(12):2301–2309, 2011.

[20] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[21] G. Burtini, S. Fazackerley, and R. Lawrence. Time Series Compression for Adaptive Chart Generation. In *CCECE '13*, pages 1–6. IEEE, 2013.

[22] MonetDB B.V. SQL Reference Manual: Temporal types, September 2016. Available online: https://www.monetdb.org/Documentation/SQLreference/Temporal.

[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, pages 28–38, December 2015.

[24] J. X. Chen and X. Wang. Approximate Line Scan-Conversion and Antialiasing. *Computer Graphics Forum*, 18(1):69–78, 1999.

[25] E. H. Chi and J. T. Riedl. An Operator Interaction Framework for Visualization Systems. In *Symposium on Information Visualization '98*, pages 63–70. IEEE, 1998.

[26] W. G. Cochran. *Sampling Techniques*. Wiley, 1977.

[27] E. F. Codd. A Data Base Sublanguage Founded on the Relational Calculus. In *ACM SIGFIDET 1971 (now SIGMOD) Workshop on Data Description, Access and Control*, pages 35–68, 1971.

[28] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, Madden S., J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[29] D. Salomon. *Data Compression*. Springer, 2007.

[30] D. H. Douglas and T. K. Peucker. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *The Cartographica Journal*, 10(2):112–122, 1973.

[31] Q. Duan, P. Wang, M. Wu, W. Wang, and S. Huang. Approximate Query on Historical Stream Data. In *DEXA '11*, pages 128–135. Springer, 2011.

[32] S. G. Eick and A. F. Karr. Visual Scalability. *Journal of Computational and Graphical Statistics*, 11(1):22–43, 2002.

[33] N. Elmqvist and J.-D. Fekete. Hierarchical Aggregation for Information Visualization: Overview, Techniques and Design Guidelines. *TVCG*, 16(3):439–454, 2010.

[34] P. Esling and C. Agon. Time-Series Data Mining. *ACM Computing Surveys*, 45(1):12–34, 2012.

[35] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database-Data Management for Modern Business Applications. *ACM SIGMOD Record*, 40(4):45–51, 2012.

[36] F. Färber, J. Dees, M. Weidner, S. Bäuerle, and W. Lehner. Towards a Web-Scale Data Management Ecosystem Demonstrated by SAP HANA. In *ICDE*, pages 1259–1267. IEEE, 2015.

[37] S. Few. *Show Me the Numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[38] E. Fink and H. S. Gandhi. Compression of Time Series by Extracting Major Extrema. *Journal of Experimental & Theoretical Artificial Intelligence*, 23(2):255–270, 2011.

[39] E. Fink, K. B. Pratt, and H. S. Gandhi. Indexing of Time Series by Major Minima and Maxima. In *IEEE International Conference on Systems, Man, and Cybernetics '03*, 2003.

[40] T. Fu. A review on time series data mining. *EAAI Journal*, 24(1):164–181, 2011.

[41] T. Fu, F. Chung, C. Lam, R. Luk, and C. Ng. Adaptive Data Delivery Framework for Financial Time Series Visualization. In *ICMB*, pages 267–273. IEEE, 2005.

[42] T. Fu, F. Chung, R. Luk, and C. Ng. Stock Time series pattern matching: Template-based vs. rule-based approaches. *EAAI Journal*, 20(3):347–364, 2007.

[43] T. Fu, F. Chung, R. Luk, and C. Ng. Representing financial time series based on data point importance. *EAAI Journal*, 21(2):277–300, 2008.

[44] S. Gandhi, L. Foschini, and S. Suri. Space-efficient Online Approximation of Time Series Data: Streams, Amnesia, and Out-of-order. In *ICDE*, pages 924–935. IEEE, 2010.

[45] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218. IEEE, 1993.

[46] The PostgreSQL Global Development Group. PostgreSQL 9.3.11 Documentation, September 2016. Available online: http://www.postgresql.org/docs/9.3.

[47] P. J. Haas. Speeding up DB2 UDB Using Sampling. Technical report, IBM Almaden Research Center, 2003. Available online: http://www.almaden.ibm.com/cs/people/peterh/idugjbig.pdf (September 2016).

[48] R. B. Haber and D. A. McNabb. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In *Visualization in Scientific Computing*, pages 74–93. IEEE, 1990.

[49] M. C. Hao, U. Dayal, D. A. Keim, and T. Schreck. Multi-Resolution Techniques for Visual Exploration of Large Time-Series Data. In *EUROVIS '07*, pages 27–34. KOPS, 2007.

[50] M. Harrower and C. A. Brewer. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[51] C. G. Healey. Choosing Effective Colours for Data Visualization. In *IEEE Visualization '96*, pages 263–270, 1996.

[52] P. S. Heckbert and M. Garland. Survey of Polygonal Surface Simplification Algorithms. Technical report, Carnegie Mellon University, 1997.

[53] J. Heer and B. Shneiderman. Interactive Dynamics for Visual Analysis. *ACM Queue*, 10(2):30, 2012.

[54] J. Hershberger and J. Snoeyink. Speeding Up the Douglas-Peucker Line-Simplification Algorithm. Technical report, University of British Columbia, 1992.

[55] D. Hilbert. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[56] M. Hilbert and P. López. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65, 2011.

[57] D. A. Huffman et al. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101. IEEE, 1952.

[58] N. Q. V. Hung, H. Jeung, and K. Aberer. An Evaluation of Model-Based Approaches to Sensor Data Compression. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2434–2447, 2013.

[59] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. P. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD '09*. ACM, 2009.

[60] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The DEBS 2012 Grand Challenge. In *DEBS '12*, pages 393–398. ACM, 2012.

[61] C. Franklin Jr. SAP lays out its internet of things platform. *InformationWeek*, May 2015. Available online: http://www.informationweek.com/strategic-cio/a/d-id/1320301 (September 2016).

[62] U. Jugel. Internal communication on machines and their sensors of a large SAP customer, September 2015.

[63] U. Jugel and V. Markl. Interactive Visualization of High-Velocity Event Streams. *PVLDB*, 5(13), 2012. (PhD Workshop).

[64] D. A. Keim. Designing Pixel-Oriented Visualization Techniques: Theory and Applications. *TVCG*, 6(1):59–78, 2000.

[65] D. A. Keim, C. Panse, J. Schneidewind, M. Sips, M. C. Hao, and U. Dayal. Pushing the Limit in Visual Data Exploration: Techniques and Applications. In *KI 2003: Advances in Artificial Intelligence*, volume 2821 of *LNCS*, pages 37–51. Springer, 2003.

[66] K. Kellenberger and S. Shaw. *Beginning T-SQL*. Apress, 3. edition, 2014.

[67] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[68] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM SIGMOD Record*, 30(2):151–162, 2001.

[69] E. J. Keogh and M. J. Pazzani. A Simple Dimensionality Reduction Technique for Fast Similarity Search in Large Time Series Databases. In *PAKDD*, pages 122–133. Springer, 2000.

[70] A. Kolesnikov. *Efficient Algorithms for Vectorization and Polygonal Approximation*. University of Joensuu, 2003.

[71] D. Laney. The Importance of 'Big Data': A Definition. Technical report, Gartner, 2012.

[72] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query Optimization by Predicate Move-Around. In *VLDB '94*, pages 96–107. VLDB Endowment, 1994.

[73] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, pages 323–334. ACM, 2009.

[74] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *TVCG*, 12(5):1245–1250, 2006.

[75] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013.

[76] W.Y. Ma, I. Bedner, G. Chang, A. Kuchinsky, and H. Zhang. A framework for adaptive content delivery in heterogeneous network environments. In *Multimedia Computing and Networking*, volume 3969, pages 86–100. SPIE, 2000.

[77] J. Mackinlay, P. Hanrahan, and C. Stolte. Show Me: Automatic Presentation for Visual Analysis. *TVCG*, 13(6):1137–1144, 2007.

[78] R. MacNicol and B. French. Sybase IQ Multiplex - Designed for Analytics. In *VLDB '04*, pages 1227–1230. VLDB Endowment, 2004.

[79] M. N. Mattos, H. Darwen, P. Cotton, P. Pistor, K. Kulkarni, S. Dessloch, and K. Zeidenstein. SQL99, SQL/MM, and SQLJ: An Overview of the SQL Standards. Technical report, IBM Database Common Technology, November 2001.

[80] K. Morton, R. Bunker, J. Mackinlay, R. Morton, and C. Stolte. Dynamic workload driven data integration in tableau. In *SIGMOD '12*, pages 807–816. ACM, 2012.

[81] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 Grand Challenge. In *DEBS '13*, pages 289–294. ACM, 2013.

[82] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[83] A. Parameswaran, N. Polyzotis, and H. Garcia-Molina. SeeDB: visualizing database queries efficiently. *PVLDB*, 7(4):325–328, 2013.

[84] D. Pineo and C. Ware. Data Visualization Optimization via Computational Modeling of Perception. *TVCG*, 18(2):309–320, 2012.

[85] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[86] K. B. Pratt and E. Fink. Search for Patterns in Compressed Time Series. *International Journal of Image and Graphics*, 2(01):89–106, 2002.

[87] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel. Recursive Query Facilities in Relational Databases: A Survey. In *Database Theory and Application, Bio-Science and Bio-Technology*, volume 118 of *CCIS*, pages 89–99. Springer, 2010.

[88] U. Ramer. An Iterative Procedure for the Polygonal Approximation of Plane Curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972.

[89] K. Reumann and A. P. M. Witkam. Optimizing Curve Segmentation in Computer Graphics. In *International Computing Symposium '74*, pages 467–472. North-Holland Publishing Company, 1974.

[90] M. Saecker and V. Markl. Big Data Analytics on Modern Hardware Architectures: A Technology Survey. In *Business Intelligence*, volume 138 of *LNBIP*, pages 125–149. Springer, 2013.

[91] W. Shi and C. Cheung. Performance Evaluation of Line Simplification Algorithms for Vector Generalization. *The Cartographic Journal*, 43(1):27–44, 2006.

[92] B. Shneiderman. Tree VisualizationWwith Tree-Maps: 2-d Space-Filling Approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

[93] B. Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE Software*, 11(6):70–77, 1994.

[94] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *IEEE Symposium on Visual Languages '96*, pages 336–343, 1996.

[95] B. Shneiderman. Extreme visualization: squeezing a billion records into a million pixels. In *SIGMOD '08*, pages 3–12. ACM, 2008.

[96] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO-DB2's learning optimizer. In *VLDB '01*, pages 19–28. VLDB Endowment, 2001.

[97] E. R. Tufte. *Beautiful Evidence*. Graphics Press, 2006.

[98] C. Upson, Thomas A. Faulhaber Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam. The Application Visualization System: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[99] R. L. Villars, C. W. Olofson, and M. Eastwood. Big Data: What It Is and Why You Should Care. Technical report, IDC, 2011.

[100] M. Visvalingam and J. Whyatt. Line generalisation by repeated elimination of points. *The Cartographic Journal*, 30(1):46–51, 1993.

[101] M. Visvalingam and J.D. Whyatt. Line Generalisation by Repeated Elimination of the Smallest Area. Technical report, University of Hull, 1992.

[102] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[103] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufman, 2000.

[104] R. Wesley, M. Eldridge, and P. T. Terlecki. An Analytic Data Engine for Visualization in Tableau. In *SIGMOD '11*, pages 1185–1194. ACM, 2011.

[105] E. Wu, L. Battle, and S. Madden. The Case for Data Visualization Management Systems. *PVLDB*, 7(10):903–906, 2014.

[106] B.-K. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary $\mathcal{L}_p$ Norms. pages 385–394. VLDB '00, VLDB Endowment, 2000.

[107] B. A. Yost. *The Visual Scalability of Integrated and Multiple View Visualizations for Large, High Resolution Displays.* Virginia Polytechnic Institute and State University, 2007.

[108] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A. Evaluation of Visual Data Analysis Tools

The following experiments were conducted in 2013 with a set of renowned visual data analysis tools, to determine how state-of-the-art business intelligence software handles datasets with $10k$ to over $1M$ records in basic charts.

Evaluation Scenario

Data Source: A time series relation $T(x, y)$, where x and y are *double* values, stored in PostgreSQL 9.3 (postgresql.org), containing the following values.

- unique random values $x \in [0, 192000]$ with a distance $dx = (x_{i-1} - x_i) \in [0.5, 1.5]$ and an average distance of $\widetilde{dx} = 1.0$
- random values $y \in [0, 75]$

Cardinality: $|T| = 1.3M$ rows

Hardware: Lenovo T510, $4GB$ RAM, Intel Core i5 M540 with $2.53GHz$, Windows 7 Enterprise ($64bit$)

Examined Tools:

1. *Tableau* Desktop 8.1 (tableausoftware.com)
2. *Datawatch* Desktop 12.2 (datawatch.com)
3. *QlikView* 11.20 (clickview.com)
4. *SAP Lumira* 1.13 (saplumira.com)

- *pgAdmin III* 1.18.1 (pgadmin.org), PostgreSQL administration tool and SQL client. Used only to measure execution times of queries for comparison.

Examination Procedure:

1. Create new worksheet.
2. Connect to data source in relational database.
3. Create a line chart of $T(x, y)$, i.e., configure the parameters of a line chart with x as an analytical dimension (column) and y as a corresponding measure (attribute).
4. Measure the time to produce a visualization, starting the clock after the configuration is finalized and stopping the clock when the visualization is perceivable.

Evaluation Results

The measured processing times of the examined data visualization tools are summarized in Table A.1. The observations are the following.

1. **Tableau Desktop** allows working on live data or import the dataset into the tableau data engine. In both cases, setting up a simple line chart is possible and Tableau requires about 30 seconds to render it. However, memory consumption becomes a problem when using too large data sets with more than $1M$ records. Moreover, the user has to be careful which visualization options to use. For instance, when setting up x as analytical *dimension* (dragging x to the list of dimensions) and using this dimension in a chart (dragging x to the chart pane), instead of visualizing x as an attribute (dragging x directly from the list of attributes to the chart pane), the tool would quickly become unresponsive, caused by the memory consumption going quickly up to $2GB$ and more.
2. **QlikView Desktop** can be connected to a database via ODBC. Therefore, a simple selection query may be added to the project file via QlikView's connection wizard. The raw data takes about 20 seconds to load. Once the data is stored in QlikView, the user can compose a line chart, which rendered in less than 10 seconds. However, QlikView ended up using at $1.3GB$ system memory, and again, choosing wrong options would quickly consume all system memory and leave the system unresponsive.

	rendering time			*limitations*
records	10k	100k	1.3M	
Tableau	$< 1s$	$< 3s$	$\sim 30s$	unresponsive at $> 1M$ records
QlikView	$< 1s$	$< 2s$	$< 10s$ ($\sim 30s$)*	becomes unresponsive over time
Datawatch	$< 1s$	$< 3s$	$\sim 30s$	unresponsive at $> 100k$ records
Lumira	$< 1s$	-	-	hard-coded limit of $10k$ records

*If considering time for raw data import as overhead.

Table A.1.: Summary of evaluation results.

3. **Datawatch Designer** allows to visualize static and streaming data, but excepts only appropriately formatted string based timestamps. A simple numeric data column was not possible to be used as a time axis for the chart. Even after converting the numeric times to appropriate timestamps, the tool was not able to visualize datasets with more than $100k$ records, without becoming unresponsive, i.e., staying at 100% CPU usage for minutes, until requiring to be stopped forcefully. As with Tableau and QlikView, selecting wrong options may easily lead to freezing the tool.

4. **SAP Lumira** quickly imported all data into the embedded Sybase IQ [78] database for further operations. Unfortunately, Lumira does not allow to produce a visualization of the raw data, since the tool rejects to visualize any internal query results containing 10,000 records or more. The user has to either limit the size of the raw dataset or define additional calculated attributes to present an aggregated version of the data instead of the raw data. However, the hard-coded limit effectively prevented freezes and crashes, when working with large datasets.

Query Execution Times

For comparison, the following queries were executed on the $1.3M$ records in the PostgreSQL database via its included database client software *pgAdmin III*.

Q_{all} A simple selection of all records, as expected to be issued by most data visualization tools.

```
SELECT x,y FROM T ORDER BY x ASC
```

$Q_{tableau}$ A naive data aggregation query, as issued by Tableau, unnecessarily grouping the records by their unique timestamp.

```
SELECT "T"."x" AS "none:x:ok",
MAX("T"."y") AS "TEMP(attr:y:qk)(1661967315)(0)",
MIN("T"."y") AS "TEMP(attr:y:qk)(4262507295)(0)"
FROM "public"."T" "T"
GROUP BY 1
```

The query was obtained from Tableau's log file (tabprotosrv.txt). If the data would contain more than one value, i.e., 3 or more values, per timestamp, this query would model a form of data reduction similar to the MinMax aggregation. However, there is no parameter to control the group size and thus the output of the query solely depends on the data distribution. As a result, Tableau reduces data volumes only for a small set of uncommon data sources and in an unpredictable way.

Q_{paa} An averaging query selecting up to $w = 1000$ average records.

```
SELECT min(x), avg(y) FROM T GROUP BY floor(x/192)
```

Note that, knowing $w = 1000$, $x_{min} = 0$ and $x_{max} = 192000$ and thus $dx = 192000$, the grouping function f_g is defined for a 1D chart as follows.

$$f_g(t) = \lfloor w \cdot (x - x_{min})/dx \rfloor = \lfloor 1000 \cdot x/192000 \rfloor = \lfloor x/192 \rfloor$$

Q_{M4} An M4 data aggregation query, selecting up to $w \approx 10000$ visually aggregated records.

```
SELECT x,y FROM (
  SELECT floor(x/192) k, min(y) y1, max(y) y2,
                         min(x) x1, max(x) x2
  FROM T GROUP BY k
) as A_m4 JOIN T
ON  A_m4.k = floor(x/192)
AND (y1 = y OR y2 = y OR x1 = x OR x2 = x)
```

Each query was run 5 times. The results are listed in Table A.2.

Discussion

All tools were able to display smaller subsets of the data with less then $10k$ records without notable latencies. However, when trying to load the entire dataset, the tools either failed completely or took at least 30 seconds before any

query	*w*	*total query answer time (ms)*					*records*
Q_{all}	-	22285	22231	21487	21327	22875	1.3M
$Q_{tableau}$	-	33365	32749	33390	31771	30189	1.3M
Q_{paa}	1000	1897	1938	1930	1870	1873	230
Q_{m4}	100	3736	3588	3525	3619	3557	162
Q_{m4}	1000	4368	4384	4337	4353	4711	915
Q_{m4}	10000	5202	5039	4649	4525	4634	8308

Table A.2.: Visualization-related SQL queries.

visualization was available; often becoming unresponsive when subsequently interacting with the visualization.

The long waiting times are caused by the naive selection of up to $1.3M$ records, which takes at least 20 seconds on the described system, of which at least 15 seconds are used for data transfer. As exemplified by Tableau's data acquisition query $Q_{tableau}$, additional query operators, introduced by the visualization tools, may cause additional processing overhead. On the contrary, a simple data reduction like the averaging query Q_{paa} and even complex visualization-driven data reduction queries like the M4 query Q_{m4} can be computed in $2-5$ seconds, producing small result sets with less than $10k$ records, transferred from the database in less than $40ms$.

The results clearly show that current visual data analysis tools are not well prepared for high-volume data sources. All data reduction tasks are imposed on the user, who needs to manually configure aggregation queries or define calculated attributes. Without such manual effort and without detailed knowledge about the visual aggregation of the considered chart types, e.g., as presented in this work, the user is not able to obtain an unfiltered view on the raw data.

B. Matrix Configuration Queries

The following SQL queries demonstrate how to compute the matrix parameters for the automatic chart matrix configuration as described in Section 6.4.2.

Listing B.1 lists and documents the steps for deriving the required subquery $Q(w_c, h_c, c_{uv})$ that can be used subsequently by a VDDA query to project a multitude of series to the 2D space of the visualization, i.e., the canvases of the subcharts of a chart matrix.

Listing B.1: SQL subqueries for computing matrix parameters inline.

```
WITH Q AS (SELECT id,t,v FROM sensors),
Q_m AS (
    SELECT count(distinct id) m       -- count data subsets
    FROM Q HAVING m > 0),             -- unless Q is empty
Q_a(m,m_p,w_min,h_min,W,H,q_wh) AS    -- define input parameters:
        (SELECT 1.0*m, 1.0*$m_p,      -- * series count and series limit
            1.0*$w_min, 1.0*$h_min,   -- * chart size limits
            1.0*$W, 1.0*$H,           -- * matrix size
            1.0*$w_min/$h_min         -- * aspect ratio
         FROM Q_m),
Q_u  AS (SELECT *, W*H/(w_min*w_min)/m m_u -- unconstrained series capacity
         FROM Q_a),                       -- (exemplary for line charts)
Q_c  AS (SELECT *, CASE                -- optimal number of matrix cells:
            WHEN m = 0      THEN 1     -- * at least one cell
            WHEN m_p <= m_u THEN m/m_p -- * limited by perceptual series limit
            ELSE m/m_u END c           -- * determined by unconstr. capacity
        FROM Q_u),
Q_s  AS (SELECT *,
            GREATEST(w_min,LEAST(W,sqrt(q_wh*W*H/c))) w_s, -- constrain w
            GREATEST(h_min,LEAST(H,sqrt(W*H/c/q_wh))) h_s  -- constrain h
         FROM Q_c),
Q_uv AS (SELECT *,                          -- compute discrete parameters:
            round(W/w_s) u, round(H/h_s) v  -- * number of cols u and rows v
         FROM Q_s),                         --
Q_p  AS (SELECT *,                          --
            floor(W/u) w_c, floor(H/v) h_c, -- * cell width w_c and height h_c
            u*v c_uv                        -- * and number of matrix cells
         FROM Q_uv)
SELECT w_c, h_c, c_uv FROM Q_p    -- expose w_c, h_c, c_uv for subsequent VDDA
```

Listing B.2 lists specific matrix configuration subqueries, with specific input parameters m_p, w_{min}, h_{min}, W, and H, including a DDL statement to create a set of test cases to simulate the matrix configuration. The query was tested in PostgreSQL 9.3.

Listing B.2: Test setup for matrix parameter computation.

```
/* 1. Create a table of test cases with varying series count. */
CREATE TABLE tests ( m int );
INSERT INTO tests VALUES
    (1),(5),(10),(20),(50),(100),(200),(500),(1000),
    (2000),(5000),(10000),(20000),(50000),(100000);

/* 2. Query the test table and compute the matrix parameters for each case. */
WITH
Q_m AS (SELECT m FROM tests),              -- use tests counts
Q_a(m,m_p,w_min,h_min,W,H,q_wh) AS         -- define input parameters:
        (SELECT 1.0*m, 1.0*20,             -- * series count and series limit
            1.0*20, 1.0*10,                -- * chart size limits
            1.0*1980, 1.0*1050, 1.0*20/10  -- * matrix size and aspect ratio
         FROM Q_m),
Q_u  AS (SELECT *, W*H/(w_min*w_min)/m m_u -- unconstrained series capacity
         FROM Q_a),                        -- (exemplary for line charts)
Q_c  AS (SELECT *, CASE                 -- optimal number of matrix cells:
            WHEN m = 0      THEN 1      -- * at least one cell
            WHEN m_p <= m_u THEN m/m_p  -- * limited by perceptual series limit
            ELSE m/m_u END c            -- * determined by unconstr. capacity
        FROM Q_u),
Q_s  AS (SELECT *,
            GREATEST(w_min,LEAST(W,sqrt(q_wh*W*H/c))) w_s, -- constrain w
            GREATEST(h_min,LEAST(H,sqrt(W*H/c/q_wh))) h_s  -- constrain h
         FROM Q_c),
Q_uv AS (SELECT *,                          -- compute discrete parameters:
            round(W/w_s) u, round(H/h_s) v  -- * number of cols u and rows v
         FROM Q_s),                         --
Q_p  AS (SELECT *,                          --
            floor(W/u) w_c, floor(H/v) h_c, -- * cell width w_c and height h_c
            u*v c_uv                        -- * and number of matrix cells
         FROM Q_uv),
Q_test AS (                                -- test output shows:
    SELECT m, W, H, w_c, h_c, c_uv,        -- * main parameters
        ceil(LEAST(m_p,w_c/w_min)) m_c,    -- * actual series capacity
        ceil(LEAST(m_p,w_c/w_min))*u*v m2  -- * actual matrix capacity
    FROM Q_p)
SELECT *, m2/m overcap,                    -- * relative capacity overhead
   1 - w_c * h_c * c_uv / W / H waste      -- * relative wasted white space
FROM Q_test
```