Hillview: A trillion-cell spreadsheet for big data

Mihai Budiu
mbudiu@vmware.com
VMware Research

Parikshit Gopalan
pgopalan@vmware.com
VMware Research

Lalith Suresh
lsuresh@vmware.com
VMware Research

Udi Wieder
uwieder@vmware.com
VMware Research

Han Kruiger
University of Utrecht

Marcos K. Aguilera
maguilera@vmware.com
VMware Research

arXiv:1907.04827v1 [cs.DC] 10 Jul 2019

ABSTRACT

Hillview is a distributed spreadsheet for browsing very large datasets that cannot be handled by a single machine. As a spreadsheet, Hillview provides a high degree of interactivity that permits data analysts to explore information quickly along many dimensions while switching visualizations on a whim. To provide the required responsiveness, Hillview introduces visualization sketches, or *vizketches*, as a simple idea to produce compact data visualizations. Vizketches combine algorithmic techniques for data summarization with computer graphics principles for efficient rendering. While simple, vizketches are effective at scaling the spreadsheet by parallelizing computation, reducing communication, providing progressive visualizations, and offering precise accuracy guarantees. Using Hillview running on eight servers, we can navigate and visualize datasets of tens of billions of rows and trillions of cells, much beyond the published capabilities of competing systems.

PVLDB Reference Format:
Mihai Budiu, Parikshit Gopalan, Lalith Suresh, Udi Wieder, Han Kruiger, and Marcos K. Aguilera. A Sample Proceedings of the VLDB Endowment Paper in LaTeX Format. *PVLDB*, 12(11): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3342263.3342279

1. INTRODUCTION

Enterprise systems store valuable data about their business. For example, retailers store data about purchased items; credit card companies, about transactions; search engines, about queries; and airlines, about flights and passengers. To understand this data, companies hire data analysts whose job is to extract deep business insights. To do that, analysts like to use spreadsheets such as Excel, Tableau, or PowerBI, which serve to explore the data *interactively*, by plotting charts, zooming in, switching charts, inspecting raw data, and repeating. Rapid interaction distinguishes spreadsheets from other solutions, such as analytics platforms and batch-based systems. Interaction is desirable, because the analyst does not know initially where to look, so she must explore data quickly along many dimensions and change visualizations on a whim.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342279

Unfortunately, enterprise data is growing dramatically, and current spreadsheets do not work with big data, because they are limited in capacity or interactivity. Centralized spreadsheets such as Excel can handle only millions of rows. More advanced tools such as Tableau can scale to larger data sets by connecting a visualization front-end to a general-purpose analytics engine in the back-end. Because the engine is general-purpose, this approach is either slow for a big data spreadsheet or complex to use as it requires users to carefully choose queries that the system is able to execute quickly. For example, Tableau can use Amazon Redshift as the analytics back-end but users must understand lengthy documentation to navigate around bad query types that are too slow to execute [17].

We propose *Hillview*, a distributed spreadsheet for big data. Hillview can navigate and visualize hundreds of columns and tens of billions of rows, totaling a trillion cells, far beyond the capability of the best interactive tools today. Hillview uses a distributed system with worker servers that provide storage and computation. It achieves massive data scalability with just a few servers (e.g., with eight commodity servers it supports a trillion spreadsheet cells).

The main challenge facing Hillview is to provide near real-time performance despite having to compute over big data.

To address this challenge, Hillview invokes a common idea in database design: specialize the engine [91]. Rather than using a general-purpose analytics engine, Hillview introduces a new engine specialized to render the tabular views and charts of a spreadsheet. The main technical novelty of the paper is how to accomplish this specialization: we introduce the notion of *visualization sketches* or simply *vizketches*, and we propose a new distributed engine to render visualizations quickly using vizketches.

Vizketches combine ideas from the algorithms and computer graphics communities. In the algorithms community, *mergeable summaries* [2] are approximate computations that compute results over disjoint subsets of the data, that can then be merged to obtain the final result. Mergeable summaries are useful to distribute the computation efficiently with fine control over the accuracy and resolution of the result. A vizketch combines mergeable summaries with a basic principle in computer graphics rendering: *compute only what you can display*. A vizketch, thus, adjusts its accuracy and resolution to match the display resolution and compute only what can be visually discerned. For example, a vizketch for producing histograms limits the number of bars to $\approx$100 and computes the height of each bar only to the nearest pixel; these choices reduce the network communication and enable computation over big data. If the user zooms in on the histogram, the vizketch adapts to the new visualization to adjust the histogram buckets and enhance the accuracy of the bars while avoiding the computation of bars that are no longer visible.

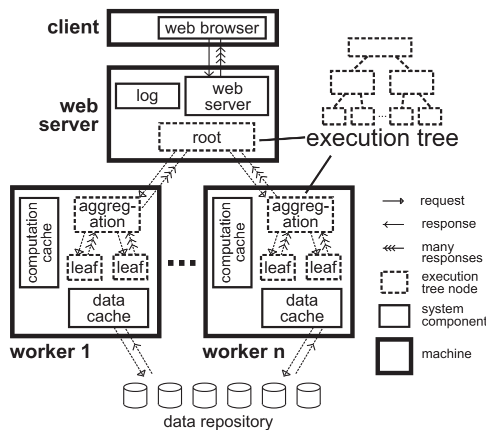


Figure 1: Hillview is a spreadsheet for browsing big data. It introduces a novel database engine based on vizketches to distribute, parallelize, and optimize the computation of visualizations and obtain interactive speeds despite large datasets. Vizketches are executed in a tree, where leafs process shards in parallel and merge results toward the root.

Vizketches play a crucial role in Hillview. They parallelize the computation, reduce communication bandwidth, enhance computation efficiency, permit a progressive visualization of results, provide a precise accuracy guarantee, and ensure scalability (§4.4). These benefits are key for a spreadsheet to be able to browse big data at interactive speeds. Furthermore, vizketches *can always* be computed efficiently. This feature differentiates Hillview from traditional visualization solutions, which let users specify broad declarative queries without exposing their performance to the user, which is problematic for efficiency or usability. That leads to an important question about Hillview: are the queries supported by vizketches rich enough to implement a fully functional spreadsheet? A contribution of this paper is to answer this question positively.

To render visualizations quickly, Hillview introduces a new distributed engine to compute vizketches (Fig.1). Clients access the system via a user interface in a web browser (top of figure), while the dataset is partitioned across a set of worker servers (bottom). The user interface triggers a visualization, such as a histogram on a chosen column. To produce the visualization, the system executes two phases: preparation and rendering. The *preparation phase* computes broad parameters required to produce a proper visualization—for example, a histogram needs to find the data range and number of items to determine appropriate bucket boundaries and sampling rates. Next, the *rendering phase* computes the values required for the visualization—for example, the height of each histogram bar. This phase utilizes a vizketch to compute with the minimum accuracy for a good visualization. The rendering phase produces partial results that incrementally update the visualizations, so the client sees an initial visualization quickly and subsequently sees more precise results. Both preparation and rendering phases use an execution tree to distribute the computation across the workers. The engine provides other important functionality that we describe in the paper: caching computations, distributed garbage-collection, and failure recovery. Furthermore, the engine has a modular design that allows developers to add visualizations easily using new vizketches without dealing with concurrency, communication, and without needing to understand the structure of an existing query optimization engine; in practice support for a new storage layer or for a new visualization type can be added in a couple of person-days of work.

The engine of Hillview differs fundamentally from general-purpose query engines in two important ways. First, due to the characteristics of vizketches, Hillview queries are scalable by construction: more specifically, queries are guaranteed to run in time $O(n/c)$, produce results of length $O(\log n)$, using memory of size $O(\log n)$ where n is the number of elements in the dataset and c is the number of worker cores[1]. In addition, many queries run in time $O(1)$. Second, Hillview produces compact results designed to be rendered efficiently on the screen. By contrast, general-purpose engines are not concerned about efficiency renderings; their queries could produce large results that take longer to visualize than to compute (e.g., returning billion points to be plotted) [17, 1, 88].

We evaluate Hillview and its vizketches. We find that Hillview can support tables with 1.4 trillion cells while providing fast response. With this scale and data in memory, operations take 1–15 seconds. Hillview displays an initial partial views even faster, which is incrementally updated until it converges to the final view. With cold data read from an SSD, operations take 2–24 seconds, and an initial view still appears within seconds. For datasets with hundreds of billions of cells, Hillview computes complete answers in under a second for most queries. This is faster than the current approach of connecting a visualization front-end to a general-purpose analytics back-end. We also find that Hillview has broad functionality for answering a wide range of questions. Vizketches are an order of magnitude faster than a popular commercial in-memory database system to compute histograms; and their performance scales linearly or sometimes super-linearly with the number of threads and servers.

To demonstrate the usability of Hillview, we provide a short video and a live demo running on AWS using small EC2 instances (these links are also available in our github repository):

Video:https://1drv.ms/v/s!AlywK8G1COQ_jeRQatBqla3tvgk4FQ

Demo:http://ec2-18-217-136-170.us-east-2.compute.amazonaws.com:8080

In summary, in this paper we propose Hillview, a spreadsheet for big data. Hillview makes two novel contributions. First, it introduces vizketches, an idea that combines mergeable summaries with visualization principles; we give vizketches for each chart and tabular view in Hillview, by finding appropriate mergeable summaries and parameters to render information efficiently yet provably accurately. Second, Hillview demonstrates how to efficiently compute vizketches by introducing a new scalable distributed analytics engine that caches computations, performs distributed garbage-collection, and handles failure recovery, while achieving the scalability and speed required for an interactive spreadsheet.

While the above contributions are pragmatic, we believe this work also contains a fundamental contribution. We raise and defend two hypotheses: (1) mergeable summaries are powerful enough to efficiently and accurately visualize massive datasets, and (2) spreadsheets can significantly benefit from a specialized engine. Hillview demonstrates these hypotheses empirically by giving vizketches for many visualizations, by building an engine for vizketches, and by quantifying its benefits. Hillview is an open-source system with an Apache 2 license, available at https://github.com/vmware/hillview.

Due to space limitations, we provide an extended version of this paper [13], with additional details: a formal computational model that captures vizketches, formal definitions of correctness and efficiency, detailed descriptions of vizketches, and correctness proofs.

[1] Assuming a balanced partition of the data between workers.

2. WHY A NEW ENGINE

In a famous paper, Stonebraker et al. advocate for designing database systems targeted for specific domains, because doing so can dramatically improve performance over one-size-fits-all solutions [91]. This approach has worked well for several domains: data warehousing, stream processing, text, scientific, online transaction processing, etc. More recently, Fisher [38] and Wu et al [100] point to the need for collaboration between visualization and data management systems. Hillview arises from these insights: we apply the database specialization approach to big data spreadsheets, where existing solutions fall short in scale and performance.

Hillview raises an important question. Data analysts may want to apply rich pipelines to data involving different frameworks, tools, and programming languages. For example, they may use a statistical package in R, then apply a machine learning algorithm in C++, followed by some hand-written scripts in python. How can Hillview integrate in this environment, given Hillview's specialized engine?

Hillview addresses this concern by adopting a versatile data layer that can connect to other tools in the pipeline. In particular, Hillview can operate directly on data stored in SQL databases, NoSQL systems, JSON files, CSV files, columnar-oriented files such as Parquet or ORC, and other big-data systems (Hadoop+Spark, Impala), without any data transformation overheads. This is because Hillview does not require data ingestion to produce indexes, or repartition data: the efficiency of vizketches permits Hillview to operate on raw data partitioned horizontally in arbitrary ways across servers: there are no requirements that partitions contain contiguous intervals or specific hash values. The only requirements of the data layout is that (1) data be horizontally partitioned ideally with approximately equal-sized partitions available to each worker, and (2) data does not change while Hillview is running[2]. The latter requirement can be met by using a data layer that provides snapshots, immutable data, or by pausing data modifications while Hillview runs. If a processing pipeline meets these requirements, then it is easy to insert Hillview into the pipeline. For example, we can connect the output of a batch-processing system to Hillview for exploration, and then output Hillview visualizations as data files or images that are processed by subsequent tools in the pipeline.

3. GOALS AND REQUIREMENTS

Our main goal is to develop a big data spreadsheet. As a data analytics tool, we are interested in functionality to explore and summarize data, such as navigation, selection, and charts. These are mostly read-only operations—our tool is for analytics exploration rather than transaction processing, data wrangling, cleaning, etc. So, we are less interested in providing interactive editing functionality, but we wish to provide ways to compute new columns from existing ones (e.g., compute a ratio of two columns). We now explain our requirements in more detail.

3.1 Why trillions of cells

Even small and medium companies can generate a trillions cells of data. These companies collect data over time from their servers, where each server might produce logs and metrics hundreds of times per minute, and a data center could have dozens of such servers. For example, 50 servers logging 100 columns at a rate of 100 rows per minute generate in a month 21.6B cells on 216M rows, or 1T cells and 10B rows in 46 months.

[2]This requirement is common in data warehousing and analytics systems.

3.2 Environment

We target an enterprise computing environment, with tens of commodity server machines in a rack hosted in a private or public cloud. We want to use as few servers as possible, as most companies cannot afford thousands of servers to run a spreadsheet.

3.3 Tabular views functionality

At first thought, it is unclear what a spreadsheet with a billion rows should do. Clearly, paging through all rows is ineffective, but analysts may wish to find patterns and then inspect individual rows.

In our experience browsing big data, we found that a spreadsheet must support at least the following functionality.

- Select data based on rich criteria to produce fewer rows (e.g., rows with timestamps in the past hour).
- Select columns to show (e.g., date and server).
- Sort by a set of columns (e.g., date first, server next).
- Aggregate duplicates and show repetition counts (e.g., selecting just date and server could create millions of repetitions: all entries produced by each server on each day).
- Search free-form text (e.g., server Gandalf) by exact match, substring, regular expressions, case sensitivity, etc.
- Move a page forward or backward.
- Scroll forward and backward using a scroll bar.
- Extract features using tools such as heavy hitters (finds most frequent elements) and Principal Component Analysis [84].

We consider whether this functionality suffices in §7.5, but we expect the list will grow over time, much like conventional spreadsheets have evolved, so we also need a flexible framework that allows us to extend the system.

3.4 Visualization functionality

We are also interested in obtaining various visualizations of columns we choose. But we face a problem with big data: graphs with billions of points can produce useless black blobs and other clutter. We want to support visualizations that can avoid this problem [86, 33], such as histograms, stacked histograms, and heat maps (Figure 2). These visualizations generalize charts, such as x-y plots and bar charts (subsumed by heat maps); and pie charts (subsumed by heavy hitters (§3.3)). We also want to extend the system with future new visualizations.

For each visualization, we want to inspect the value of individual points, change parameters (e.g., # buckets in histogram) and, if applicable, understand trends, correlations, and swap axes. Furthermore, we want to zoom in parts of the data, by regenerating the visualization for a subset of its data as determined by a mouse selection.

3.5 Other features

Data types. We want to support integers, floating-point numbers, dates, free-form text, and strings describing categorical data.

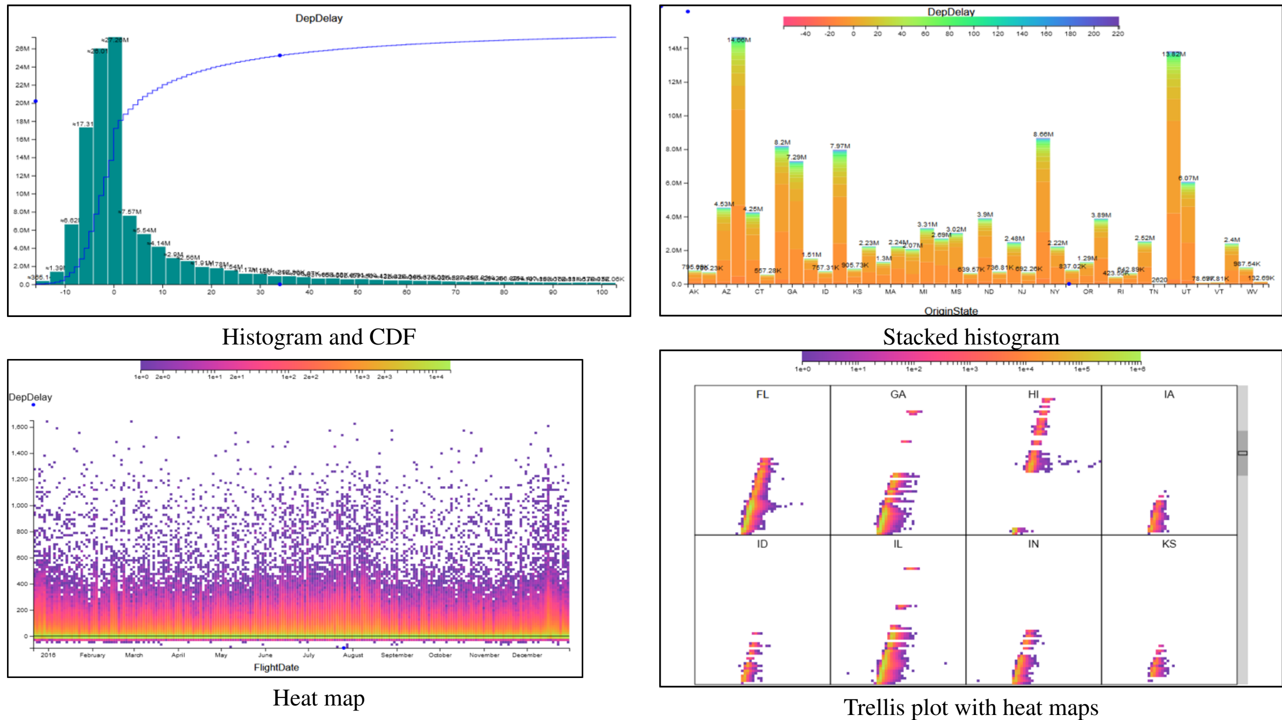
Map functions. We want to produce a new column by combining existing ones using user-defined map functions (e.g., a ratio of two columns).

Data sources. We want to read data from a variety of common sources (comma-separated files, SQL databases, row columnar files such as ORC, future formats, etc).

4. VIZKETCHES

Key to providing the required performance of Hillview, vizketches are a simple concept that combine the idea of mergeable summaries (or sketches) from the algorithms community with

Name	What it shows	Example
CDF	Distribution of variable	# events before noon
Histogram	Frequency of variable for each bucket	# events per hour of day
Stacked histogram	Frequency of first variable and frequency of second variable grouped by first	# events of each type per hour of day
Normalized stacked hist.	Ditto but bars normalized	% of events of each type per hour of day
Heat map	Frequency of two variables	# events for each server and hour
Trellis plots	Arrays of the other plots grouped by one or two variables	# events for each server and hour, for each datacenter



Histogram and CDF

Stacked histogram

Heat map

Trellis plot with heat maps

Figure 2: Some clutter-free visualizations for large datasets. Visualizations cover a single variable (column) or multiple variables, up to four.

the principle of visualization-driven computation from the graphics community.

4.1 Background

Mergeable summaries. Intuitively, a summarization method computes a compact representation ("summary") of a large dataset, which can then answer approximate queries on the dataset. A summarization method is mergeable [2] if the summary can be obtained by merging many summaries computed independently over parts of the dataset. More precisely, a mergeable summarization method consists of two functions $summarize(D)$ and $merge(S,S')$. The first takes a dataset D and returns a summary; the second merges two summaries and returns another summary. A summary is small compared to D—typically by many orders of magnitude—and it can approximate queries on D (the allowable queries depend on the choice of summarization method). Summaries of two separate datasets can be merged via the *merge* function:

$$summarize(D_1 \uplus D_2) = merge(summarize(D_1), summarize(D_2))$$

where D_1 and D_2 are mutisets and $\uplus$ is multiset union. There are summarization methods for many types of queries, such as histograms, heavy hitters, heat maps, and PCA. Many summarization methods are sketches from the streaming algorithms literature [21], and so the community sometimes mixes these two concepts. However, a summarization method can also use sampling, which can be more efficient because it does not scan all data. The summarization method has two accuracy parameters: an error ε and an error probability δ, with the guarantee that an approximation computed from a summary has error at most ε with probability $1-\delta$. For a more formal description of our computational model, we refer the reader to Appendix A of the extended version of this paper [13].

Visualization-driven computation. In computer graphics, rendering is an expensive operation that must be optimized. To do that, a basic principle is to drive the computation based on what will be visualized and its resolution, taking into consideration the limits of human perception and the lossy channels of displays. This principle is behind many graphics techniques, such as ray tracing, culling, and imposters [48].

4.2 Basic idea

A vizketch is a mergeable summary designed to produce a good visualization. More precisely, a vizketch method targets a specific visualization (e.g., a histogram) with a given display dimension (width and height in pixels). The vizketch method consists of the

two functions of a mergeable summary, *summarize* and *merge*, with parameters carefully chosen to achieve two goals: the summary is *small*, and it permits a *good rendering* of the visualization.

Small summary means that its size depends only on the length of the description of the visualization, not on the input size. More precisely, visualizations are inherently limited by the finiteness of their renderings, so they have a short description (e.g., a histogram is described by its bucket boundaries and heights). The length of this description is a lower bound on the size of the summary. We seek summaries whose size is polynomial in this length, rather than the data set size. The key hypothesis behind Hillview is that visualizations always admit vizketches with such small summaries. This hypothesis is not obvious; it can be formalized with proper definitions of the computational model, visualizations, etc., but this is outside the scope of this paper. Instead, Hillview supports this hypothesis empirically: we give vizketches for many visualizations, by adapting techniques from the sketching/streaming literature.

Good rendering means two things. First, the rendering has a bounded error with high probability (e.g., histogram bars are off by at most 1 pixel). Second, the rendering is not cluttered (e.g., there are at most 50 buckets for a histogram when the screen width is 200 pixels). The precise requirements are carefully chosen for each type of visualization. These choices are made so that a human can consume the information effectively without perceiving any errors in the approximation.

To use vizketches, Hillview defines a computation tree whose nodes are assigned to the servers (Figure 1). Hillview assumes that the data is stored on a distributed storage layer, and is sharded into small chunks, which are distributed to the tree leaves. The sharding can be arbitrary: chunks need not be sorted or partitioned by a specific key.

To perform a visualization, each leaf independently runs the vizketch's *summarize* function on the shards that it has; this function might choose to sample or scan the data in the chunk[3]. The summaries are then merged along the computation tree, using the vizketch's *merge* function. The root receives the final summary, which reflects a view of the entire dataset and produces the rendering of the visualization for the client.

Vizketches parallelize the computation across threads and servers, while reducing computation and network bandwidth to only what is necessary for a good rendering. They can also provide partial results for progressive visualizations, in addition to other benefits (§4.4). We now describe specific vizketches.

4.3 Algorithms

Hillview uses a large number of vizketches. Some produce graphs (histograms, stacked histograms, heat maps, trellis plot); others produce information for the spreadsheet tabular view (next items, quantile for scroll bar, find text, heavy hitters). We describe a few here; others are omitted due to space limitations but they follow a similar approach and can be found in Appendix B of [13]. Vizketches have rigorous guarantees of correctness, which we present in Appendix C of [13].

A vizketch is parameterized by the target display resolution, and produces calculations that are just precise enough to render at that resolution.

Histograms. We are given a numerical column (or a value that can be readily converted to a real number, such as a date) with range $[x_0, x_1)$, a number B of histogram bars, and their maximum pixel height V. The histogram vizketch (Figure 13(b)) divides the range $[x_0, x_1)$ into B equi-sized intervals, one per bin. To maximize use of screen, we should scale the bars so that the largest one has V pixels. We furthermore allow an error of .5 pixels in the estimation of the height of each bar. We prove in Appendix C of [13] that to obtain this error with probability $< 1-\delta$, it is sufficient to sample $n = O(V^2 B^2 \log(1/\delta))$ items from the dataset. Notice that this function is independent on the dataset size, and depends only on the screen size. The *summarize* function outputs a vector of B bin counts, and the *merge* function adds two vectors.

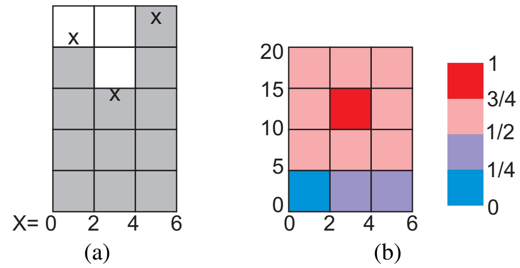


Figure 3: Charts in Hillview have an error of at most 1/2 pixel or one color shade with high probability. (a) A histogram with three bars. The × indicates the correct height for the bar at most one 1/2 pixel away from the rendering. (b) A heat map (left) and the density color map (right). The x-axis has bins for the first variable; the y-axis, for the second variable. The color indicates the density of each bin, where the error is at most one color shade with high probability.

Heat map. We are given two columns X and Y with ranges $[x_0, x_1)$ and $[y_0, y_1)$, and the pixel dimensions $H \times V$. A heat map (Figure 13(d)) defines bins in two dimensions, where each bin consumes $b \times b$ pixels (b is 2 or 3, depending on the screen resolution). Thus, we have $B_x = H/b$ and $B_y = V/b$ bins for X and Y. The density of the data in a bin is represented by a color scale. If we use $c \approx 20$ distinct colors, the required accuracy for each bin density is $1/2c$. This requires a target sample size $n = O(c^2 B_x^2 B_y^2 \log(1/\delta))$[4]. The *summarize* function samples data with the target rate, counting the number of values that fall in each bin. It outputs a matrix of $B_x \times B_y$ bin counts. The *merge* function adds two such matrices.

Next items. This vizketch is used to render a tabular view of the spreadsheet given the current row shown at the top R (or $R = \perp$ to choose the beginning of the dataset). We are also given a column sort order, and the number K of rows to show. This vizketch returns the contents of the K distinct rows that follow R in the sort order. The *summarize* function scans the dataset and keeps a priority heap with the K next values following row R in the sort order. The *merge* function combines the two priority heaps by selecting the smallest K elements and dropping the rest.

Heavy hitters. A vizketch to find heavy hitters works by sampling. Let K be the maximum number of heavy hitters desired. The basic idea is to sample with a target size n (determined below), and select an item as a heavy hitter if it occurs with frequency at least $3n/4K$. A statistical calculation shows that by picking $n = K^2 \log(K/\delta)$, with probability $1-\delta$ we can obtain all elements that occur more than $1/K$ of the time and no elements that occur fewer than $1/4K$ of the time. This method is particularly efficient if K is small. We employ several other algorithm for finding heavy hitters, described in Appendix C of [13].

[3]This choice can be made independently for each chunk.

[4]Sampling can be used only if the map from count to color is linear.

4.4 Benefits

Vizketches bring many benefits to Hillview. In the list below, the parentheses indicate from where the benefit is inherited: S means sketches/mergeable summaries, V means visualization-driven computation, and S+V means the combination of both.

- *Parallel computation (S).* Servers and cores within servers independently compute on different parts of the data, and the result is merged.
- *Bandwidth efficiency (S+V).* When a server finishes its computation, it communicates only a compact summary to be merged.
- *Computation efficiency (S+V).* Some computations can done over a small sample of data based on the required accuracy.
- *Progressive visualization (S).* As servers complete their computation, the system computes a partial summary that gradually progresses to the final result. This ensures that slow servers and tail latencies do not hinder interactivity. Users can cancel a visualization after seeing partial results.
- *Accurate visualization (S+V).* The resulting visualization has a precise accuracy guarantee.
- *Scalability (S+V).* As we add more data, vizketches can sample more aggressively to enhance efficiency while achieving the required accuracy.
- *Easy to obtain (S).* There is a rich literature on mergeable summarization methods and sketches of various types (histograms, heat maps, heavy hitters, etc); these sketches can often be converted into vizketches through a relatively simple analysis that translates the accuracy of the sketch into the required accuracy of the visualization, as illustrated above.
- *Modularity (S).* New visualizations can be added to Hillview by defining new vizketches as two simple functions (§4.1) without the developer worrying about distributed systems aspects.

5. DESIGN AND ARCHITECTURE

We now explain in detail the design and architecture of Hillview, starting with its high-level design choices (§5.1), followed by a detailed description in the subsequent sections.

5.1 Design choices

We now explain the key design choices of Hillview, which derive from the power and characteristics of vizketches.

- *Distribute computation while minimizing server coordination.* To answer a query, Hillview launches a computation tree to efficiently distribute the query to worker servers and aggregate the results according to the vizketch computations.
- *Storage-independence.* Hillview can access data in a wide variety of formats (SQL, NoSQL, text, JSON, etc), with few restrictions on how data is partitioned (§2), and without the need to pre-compute indexes or perform extract-transform-load. As a result, Hillview does not require any pre-processing to ingest data. This is beneficial to integrate Hillview into a diverse analytics pipeline (as explained in §2), and this is possible because the efficiency and parallelization of vizketches permits forgoing data conversions, repartitioning, and pre-computations.
- *Sample data in a controlled manner.* Sampling improves efficiency but introduces error. Vizketches allow Hillview to sample while bounding the error to what we can perceive.
- *Modular algorithms.* Programmers who write vizketch algorithms do not have to worry about concurrency, communication, or fault-tolerance; they just implement single-threaded code, and the architecture handles all such issues in a uniform and transparent manner.

5.2 Architecture

Figure 1 shows the architecture of Hillview. Hillview is designed as a cloud service accessible to clients through a web interface. A web browser handles user interaction with the spreadsheet and renders the the charts incrementally as computation results arrive. To produce a visualization, a web server launches the required computation as one or more *execution trees*. Communication happens only along the edges of the tree, and is restricted to small messages: queries in one direction and summaries in the other. Each tree is rooted at the web server, followed by one or more layers of aggregation nodes, and several leaf nodes. The leaf nodes perform the actual computation over disjoint partitions of the dataset. These nodes have an in-memory data cache in front of the data in repositories. There is also a computation cache to reuse prior computations. The aggregation nodes are intended to scale the system to many servers; a small deployment with tens of servers needs only one layer.

5.3 Execution tree

A visualization typically involves two execution trees, each intrinsically linked to a mergeable summary. The first tree computes data-wide parameters such as the size and range of the data set; this computation may be cached from previous visualizations. The second tree computes a vizketch for the visualization with the required accuracy based on the results produced by the first execution tree.

The execution of each tree is based on the *summarize* and *merge* functions (§4.2) of the mergeable summary. A tree executes in two phases.

The first phase initiates the computation from the root down the tree to each leaf, and causes the leaf nodes to apply the *summarize* method on their data partition. To parallelize execution within a server, each server runs multiple leaf nodes: there is a thread pool that serves leafs with work to do. To facilitate this process, the data partition within a server is divided into micropartitions of 10-20M rows, each micropartition assigned to a leaf.

The second phase, in its most basic form, executes from the leafs toward the root, causing each node to aggregate results from its children through the *merge* method. Thus, ultimately the root node combines the output of all nodes, and the result can be rendered. When processing large datasets in a distributed system, there may be variation in the processing times across servers and partitions. If the root had to wait for all other nodes to finish, its completion would be disrupted by any stragglers, affecting the interactive experience of users. To address this problem, nodes periodically propagate partially merged results of the vizketch without waiting for all children to respond. Thus, the root receives partial results and sends them to the client UI, before it gets the final results. The web browser then renders results as they arrive, so that users can see a progression of the computation. Hillview shows a progress bar that reflects the number of leafs that have completed. Users can cancel the computation based on the partial results they see.

There is a trade-off between the freshness of the partial results and the bandwidth savings produced by aggregating partial results. After receiving a result from a child node, aggregation nodes wait for 0.1 seconds and aggregate all results that arrive within this interval. This provides frequent updates to the UI; the increase in communication costs is modest because all vizketch results are small by construction.

Hillview allows users to cancel computations (e.g., because a partial visualization is satisfactory). This is done by interrupting

an execution tree with a high priority cancellation message that bypasses the queuing mechanisms in the communication between tree nodes. This message causes tree nodes to do two things: remove work for that computation that was previously enqueued, and ignore requests for micropartitions not yet started. We currently do not stop ongoing computations on a micropartition.

5.4 Data input, caching, and data output

Unlike most database systems, Hillview reads data repositories without pre-processing, repartitioning, or other optimizations. This is possible because the computational engine of Hillview—based on vizketches—makes few assumptions about the data. The assumptions are that repositories do not change while they are accessed (this can be provided by using storage snapshots or controlling write access) and data is horizontally partitioned, ideally with approximately equal-sized partitions available to each worker, so that data can be read in parallel. When a worker needs a column, it reads it completely from the data repository taking advantage of fast sequential access and columnar access if the repository supports it (SQL, Parquet, ORC). Once data is read, it is kept in an in-memory cache; the cache purges entries not used for a while (currently 2 hours).

Hillview uses two types of caching: data and computation. The first is an in-memory cache of the raw data in the data repositories. The data cache is organized by column to provide data locality, since vizketches tend to operate on relatively few columns.

The computation cache stores results produced by mergeable summaries; these results are small, allowing a large number of results to be cached. This is useful for mergeable summaries that provide auxiliary functionality, such as column statistics, which are used repeatedly and are deterministic. The computation cache is indexed by what mergeable summary was used and what dataset was operated on.

Hillview can save a derived table (§5.6) to a data repository, by having each worker store its partition of the data. This is implemented through a special vizketch with a summarize function that writes a data record to the repository and returns an error indication, while the merge function combines error indications.

5.5 Vizketch modularity and extensibility

The inherent structure of vizketches permits Hillview to cleanly separate them from the rest of its architecture so that developers can implement new vizketches without the hard concerns of distributed systems (communication, coordination, fault tolerance, etc) or data storage. Specifically, to support a new vizketch, a developer needs to implement the following things: (1) a serializable[5] type for the vizketch summary, (2) an implementation of the *summarize* and *merge* functions of the vizketch; these all operate on the in-memory columnar representation of the data, and are independent on the storage layer, (3) code to render the vizketch summary as a visualization in the user interface of the spreadsheet in the browser, (4) code to trigger the visualization through a user interface action, and (5) a function to connect the user interface action to the invocation of the vizketch in the root node. None of these functions are concerned with concurrency (they are single-threaded), and most of them can be implemented with only tens of lines of code—the sole exception is (3), which requires more code to provide the graphical functionality. We quantify the effort to for step (2) in §7.4.

[5]I.e., the type should have a serialization method to convert an instance into a byte sequence for network transmission.

5.6 Data transformations

Users may wish to generate new data from existing data as part of the data exploration process. Users can do that externally to Hillview through other analytics tools, and then import the results into Hillview for inspection (§2). Alternatively, Hillview provides some support for deriving new data through two common operations: selection (filtering) and user-defined map operations (§3).

Selection permits a user to create a new table that contains a subset of the rows of another table (e.g., rows where the year column is 2019). A particularly useful selection operation in a spreadsheet is to zoom in part of a graph, which corresponds to choosing the rows within the zoom window. To provide this functionality, Hillview allows mergeable summaries to work on subsets of rows of the dataset. More precisely, a table can be derived from other tables by choosing a subset of the rows. To save space, the tables share common data and store a "membership set" data structure that identifies which rows are contained in the table. This membership set data structure has different implementations, depending on the density of the filtered data. Dense tables that contain most rows store a bitmap, while sparse tables store a hashset of the rows indexes. This information is kept locally for each data partition.

When executing the summarize method, some vizketches work by sampling a subset of rows. We must ensure that sampling is efficient (it does not require reading each row) but it is also correct (it picks rows uniformly at random). For sparse tables, we generate the first sample by choosing a random row number for the first element; we generate the following samples by returning the next elements in sorted order of their hash values. For dense tables we walk randomly the bitmap in increasing index order.

User-defined maps permits a user to create a column from existing ones (e.g., add two columns), where the map is an arbitrary function. Some map functions are built-in (e.g., converting strings to integers); additional functions can be written by users in Javascript. To support this functionality, Hillview creates a new table with the new column populated by running the map function at the leafs of the execution tree. Currently, this data is stored only in the in-memory caches; if the cached data is reclaimed, the column is recomputed on demand. We believe this is a reasonable choice for a spreadsheet, since derived columns tend to be short-lived.

5.7 Memory management

Early versions of Hillview used a distributed garbage-collection protocol to handle memory management. This protocol was complex and fragile (for example, loss of network messages could cause memory leaks). In the current version we have simplified memory management by aggressively using only soft state: all in-memory data structures are disposable, including at leaf-, aggregation- and root nodes. The only requirement to implement this architecture is for the storage layer to provide an API to read a particular snapshot of a dataset; in this way, in-memory data is reconstructed by reloading the original snapshot. We use the Partitioned Data Set architecture from Sketch [14] to represent distributed objects; unlike sketch, all remote references are "soft" — they may not point to valid data structures.

Each machine performs independently garbage-collection; a caching layer maintains a working set of recently accessed objects in memory. In-memory cached objects at leaf nodes can be reconstructed by reading data from disk; tables obtained from filtering (§5.6) or by deriving new columns (§5.6) can be regenerated by re-executing the operation that created them in the first place.

When the root node attempts to access a remote object on a leaf which no longer exists the leaf reports an error. The root node then re-executes the query that produced the missing object. This may

require re-executing other queries, that produced the source objects; the recursion ends when data is read from disk.

To enable query re-execution, the root node maintains a redo log with all executed operations. The redo log is the only persistent data structure maintained by Hillview (recall that the storage layer is not part of Hillview).

5.8 Fault tolerance

Hillview provides fault tolerance by logging operations that initiate each execution tree, and lazily replaying operations to reconstruct node state. When the root node restarts after a failure, it reads the redo log to memory, but does not replay it yet. Replaying occurs only when the user tries to access a dataset that no longer exists, as described in §5.7.

Worker nodes are stateless, so restarting the node after a failure is equivalent to deleting all cached datasets. These datasets are reconstructed by the root node on demand by replaying log operations.

This lazy aproach is suitable for a spreadsheet, because most views are short-lived results that a user never accesses again.

For this replay mechanism to work, vizketches must be deterministic, otherwise a restarted node becomes inconsistent with nodes that never crashed. To provide determinism for randomized vizketches (e.g., those that use sampling), the log includes the seed used for randomization.

6. IMPLEMENTATION

Hillview consists of 35000 lines of Java and 16000 lines of TypeScript code. The user interface in the browser is implemented in TypeScript [95], using parts of the D3 JavaScript library [11]. Graphics is done using SVG [25]. The web server runs the Apache Tomcat application server [4]. The browser gets progressive replies from web server using a streaming RPC based on Web Sockets [37]; these RPC messages are serialized as JSON. The cloud service is implemented in Java. We use Java's type-safe object serialization facilities for sending queries and data between machines. We use the fast collections Java library [34] for efficient data structures, with customizations for faster sampling. For server-side JavaScript we use Oracle Nashorn [73].

We use a variety of open-source libraries to interface with external storage layers (e.g., csv files, various log formats (e.g., RFC 5424), JDBC connectors, columnar binary formats such as Parquet or ORC, etc). The communication between back-end machines uses GRPC [44]. The core communication APIs are based on reactive streams, using RxJava [80, 64]. We use RxJava's `Observable` datatype for many purposes: (1) It represents a stream of partial results, (2) it offers support for operation cancellation, through its `unsubscribe` method, (3) it is used for reporting progress to the user for long-running operations, displayed in the form of progress bars, and (4) it is responsible for managing concurrent execution on multi-core machines (using the `observeOn(threadPool)`; this thread pool is used for all of the workers' computations. The in-memory tables use as much as possible Java arrays of base types to reduce pressure on the Java GC. String columns use dictionary encoding for compression.

7. EVALUATION

Our evaluation goal is to determine whether Hillview provides interactive performance with large data sets, how Hillview compares to existing systems, how vizketches contribute to that goal, and how effective the spreadsheet is.

Summary. We find the following results:

- Hillview can handle spreadsheets with 130B rows and 1.4T cells using only 8 servers. At the upper range, visualizations can take 20s when loading from disk, but the first partial visualization appears in a few seconds and gets gradually updated. This is much better than existing systems (§7.1). For smaller datasets most response times are on the order of hundreds of milliseconds.
- Vizketches perform well on a single thread and scale well with the number of threads and servers. Vizketches based on sampling scale super-linearly. This performs signficantly better than a database system (§7.2).
- Vizketches are key in Hillview: they implement a broad range of functionality of the spreadsheet, to the extent that they are the sole way to access data in the system (§7.3).
- Vizketches are easy to code and do not require an understanding of distributed systems (§7.4).
- Hillview is a spreadsheet with many useful features, able to answer a diverse set of queries effectively (§7.5).

Testbed. Our testbed consists of eight servers running Linux kernel 4.4. Each server has two sockets with 14-core 2.2Ghz Intel Xeon Gold 5120 CPUs, 192 GB of DRAM, two SSDs with 381GB and 1.8TB, connected to a 10 Gbps network. The client web browser runs on a laptop connected to the servers via a 100Mbps network with 1ms ping time to the servers. This setup represents a typical enterprise setting.

Dataset. We use a dataset with US airline flight performance metrics for the past 20 years [71]. Each row has a flight with its origin, destination, flight time, departure and arrival delays, etc. This is a real dataset with numerical, categorical, text, and undefined values. There are 130 million rows and 110 columns, which amount of 58.2 GB of uncompressed data. In some experiments, we scale the dataset by a factor of 5, 10, or 100, by replicating its rows and reading them repeatedly from disk. These datasets are labeled "Flight-Kx" where $K{=}1,5,10,100$ indicates the replication factor ($K{=}1$ is the original dataset).

7.1 Hillview end-to-end performance

We measure the end-to-end time that Hillview takes to execute spreadsheet operations for datasets of various sizes.

Baseline. We compare Hillview against the traditional approach for big-data spreadsheets, such as Tableau, which is to connect a visualization front-end to a general-purpose analytics back-end. Our baseline uses Spark as the back-end, and we measure only the analytics delay (not the rendering delay), giving an advantage to the baseline. We optimize Spark to our best ability. We write queries in Scala; we pre-load all data to RAM before measuring; and we use the same optimizations for each query as Hillview, including sampling.

Workload. Figure 4 shows the visualizations we are measuring. We picked these operations using two criteria: (1) Each group of operations corresponds to a user action in the spreadsheet (e.g., ask for a histogram, or change the sort order of a tabular view). (2) The operations covers a broad range of the vizketches available in Hillview.

Setup. In each experiment, we pick an operation, a dataset size, and a system. The dataset sizes vary from 5x–100x the original data, corresponding to 650M–13B rows of data with 110 columns each, for a total of 71B–1.4T cells. We submit the operation to the system and measure its response time and amount of data received by the root node. For Hillview, we submit the operation at the user

Name	Description
O1	Sort, numerical data
O2	Sort 5 columns, numerical data
O3	Sort, string data
O4	Quantile + sort, 5 columns, numerical data
O5	Range + (histogram & cdf), numerical data
O6	Filter + range + (histogram & cdf), numerical data
O7	Distinct + range + histogram, string data
O8	Heavy hitters sampling, string data
O9	Distinct count, numerical data
O10	Range + (stacked histogram & cdf),numerical data
O11	Heatmap, numerical data

Figure 4: Spreadsheet operations. The + indicates serial operations, while & indicates concurrent operation. Numerical data refers to integer or floating point.

interface of the web browser, and we measure two response times at the browser: first partial visualization and final visualization. For the Spark baseline, we start the measurement when the computation starts, and end the measurement when the query result is obtained. For Hillview, we consider two cases: data is in memory before the measurement, and data is cold on disk (SSD). For Spark, we only consider the case with data in memory.

Results. Figure 5 shows the results for warm data in memory. We could not run Spark with a dataset larger than 5x because it exhausted the memory at the servers: for example, the 10x dataset has 582 GB on-disk but its in-memory representation expands beyond the available aggregate memory in the testbed.

The top graph shows the response time. We see that for most operations, Hillview performs at least as well as Spark, even when Hillview processes twice the data. We also see that Hillview at 100x can be slow to compute all results: 7.3–15.2s. However, Hillview produces a partial visualization quickly, which provides a better interactive experience.

The bottom graph shows the amount of data received over the network by the root node (for Hillview) or the master (for Spark); note that the Y axis is log-scale. Spark consumes an order of magnitude more bandwidth than Hillview, except for O11. This is because Hillview transmits a small amount of data to produce the visualizations. The exception, O11, is a heatmap, which contains a large number of cells and hence its vizketch carries considerable more data. We also see that Hillview consumes more bandwidth with a larger dataset. This is because the larger dataset takes longer to complete, and so Hillview transmits partial visualizations; with O11, the total amount of data becomes larger than Spark, but it is still reasonable at 3.5MB.

Figure 6 show the results for cold data on disk. For 5x and 10x data, visualizations still complete in 3s. For 100x, the delay can be 24s; first visualizations arrive earlier, often within 2.5s (not shown).

In all cases, Hillview provides acceptable performance for interaction. In our experience using Hillview, we tend to spend significantly more time browsing and analyzing charts than waiting for visualizations (cf §7.5).

7.2 Vizketch microbenchmark

We now consider the base performance of vizketches on one thread, and its scalability over threads and servers. We run each measurement multiple times, and we display the variance of measurements after excluding the fastest and slowest measurements; the variance tends to be small[6].

[6]The first measurement warms up the Java JIT compiler, so it is generally much slower.

Workload. We benchmark two types of histograms vizketches: one based on sampling (approximate, with bounded error) and the other based on streaming (no error). We run these on numeric data.

7.2.1 *Single thread performance*

Baseline. The baseline is a common high-end commercial in-memory database system performing a histogram calculation; we are not allowed to reveal its name.

Setup. In each experiment, we pick a computation method (streaming, sampling, or database system). We measure the time it takes to execute the method on a single thread on 100 million rows. For vizketches, we use a tree with a single leaf directly connected to the root, limiting execution to a single thread. For the database system, we do not constrain the number of threads that it uses.

Results. We obtain the following measurements:

Method	Time (ms)
streaming	527
sampling	197
database system	5,830

We see that the database system is an order of magnitude worse, because it has overheads that vizketches avoid: data structures must support indexes, transactions, integrity constraints, logging, queries of many types, etc. (although none of these are necessary in our case). In contrast, vizketches are specialized to perform only the required computation.

7.2.2 *Scalability to multiple CPUs*

We now consider the performance of vizketches as we run them on multiple CPUs.

Setup. We consider a computation tree that has n leafs on the same server, connected to a single root. The system executes each leaf on a separate thread, up to the available CPUs in the system. In each experiment, we pick a number n. As we increase n, we also increase the number of rows to be processed by adding more shards to the system, keeping constant the number of rows that each leaf gets—thus, the total number of leafs and the work increase together as n grows. We expect an approximately constant running time.

Results. Figure 7 shows the results. For the streaming histogram vizketch, we can see that latency remains constant up to 16 shards, showing a nearly ideal scalability up to that point. After that, the server relies on hyper-threading, which impairs scalability. For the (sampled) histogram vizketch, scalability is super-linear, because the sample size to obtain a given level of accuracy remains the same irrespective of the dataset size (§B.1). Thus, as we add more leafs, we decrease the number of samples (and work done) per leaf.

7.2.3 *Scalability to multiple servers*

Next, we consider the performance of vizketches as we run them on many servers.

Setup. In each experiment, we pick a number n of servers and a vizketch. We use a computation tree that has 64 leaf nodes on each server, connected to the root. As we increase n, we increase the number of rows by adding more shards, so that each leaf node maintains the same number of shares (and rows). We measure the time it takes to execute the vizketch running across the servers.

Results. Figure 8 shows the result. As before, for the streaming histogram vizketch, the latency remains constant as we add more servers and data, showing ideal scalability. For the sampled histogram vizketch, we again observe super-linear scalability due to

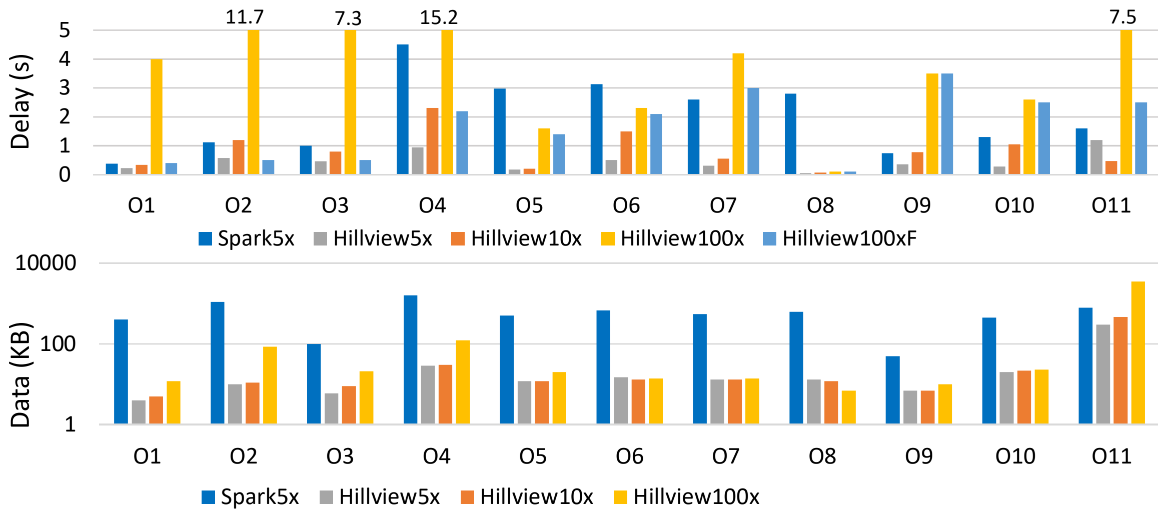


Figure 5: End-to-end performance comparison. The top graph shows the response time to produce each visualization, while the bottom graph shows how many bytes the root node received. Here, we ensure the data is in memory before the measurement starts. The bars are labeled with the system name (Spark or Hillview) and the dataset size (5x to 100x corresponding to 650M to 13B rows). Hillview100xF is the time it takes for Hillview to produce the first partial visualization running with 100x data.

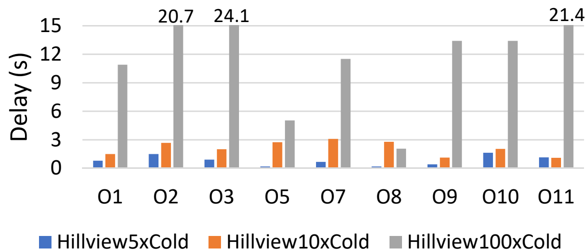


Figure 6: End-to-end performance of Hillview when data is not in memory, so it needs to be loaded from SSD. Not shown are first visualizations, which arrive within 2.5s most of the time, and within 4s always. O4 and O6 are omitted because in the spreadsheet these operations never happen with cold data (a prior action loads the data).

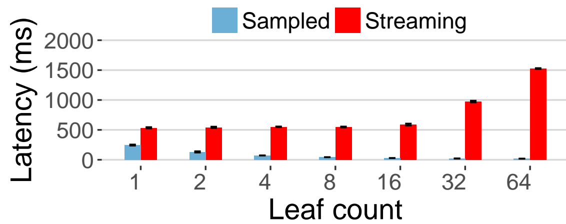


Figure 7: Scalability of vizketches as we add more leafs and shards together. Ideal scalability would be constant latency.

the same effect: the sample size remains constant, so the amount of work per server decreases with the number of servers.

7.3 Vizketch applicability

We consider our experience of using vizketch to implement the various spreadsheet functionality, to gain an understanding of the applicability of vizketches to processing data in Hillview.

When we started the project, we did not know if vizketches would suffice or we would need more powerful computation mechanisms. In building the system, however, we found vizketches to be powerful and capable of implementing a broad range of functionality: tabular views, scrolling, simple data transformations, filtering, table summaries, and various visualizations. We eventually realized that we could implement all data visualization functionality of Hillview using vizketches; in fact, Hillview has no other way to visualize data other than vizketches.

Figure 8: Scalability as we add more servers and increase the dataset proportionally. As before, the ideal scalability corresponds to a constant latency. Note that the Y axis is logarithmic.

Vizketch	LOC	Vizketch	LOC
Histogram	114	Next items	191
CDF	114	Find text	108
Stacked histogram	130	Heavy hitters (sampling)	35
Heatmap	130	Range	156
Heatmap trellis	127	Number distinct	117
Quantile	79		

Figure 9: Effort required to implement vizketches.

7.4 Vizketch coding effort

We now turn our attention to the effort required to write vizketches. We again report on our experience with Hillview.

Quantitatively, Figure 9 shows the number of lines of back-end (Java) code required to implement each vizketch in Hillview. We

Question	Description
Q1	Who has more late flights, UA or AA?
Q2	Which airline has the least departure time delay?
Q3	What is the typical delay of AA flight 11?
Q4	How many flights leave NY each day?
Q5	Is it better to fly from SFO to JFK or EWR?
Q6	How many destinations have direct flights from both SFO and SJC?
Q7	What is the best hour of the day to fly?
Q8	Which state has the worst departure delay?
Q9	Which airline has the most flight cancellations?
Q10	Which date had the most flights?
Q11	What is the longest flight in distance?
Q12	Is there a significant difference between taxi times of UA or AA on the same airport?
Q13	Which city has the best and worst weather delays?
Q14	Which airlines fly to Hawaii?
Q15	Which Hawaii airport has the best departure delays?
Q16	How many flights per day are there between LAX and SFO?
Q17	Which weekday has the least delay flying from ORD to EWR?
Q18	Which day in December has the most and least flights?
Q19	How many airlines stopped flying within the dataset period?
Q20	How many flights took off but never landed?

Figure 10: Questions used to evaluate the effectiveness of Hillview at extracting information from data.

can see that the code is compact: the largest vizketch takes only 191 lines of code. We found that an expert takes only a few hours to implement and test the code. However, some vizketches involve fairly sophisticated algorithms; selecting or developing those algorithms took considerably longer than implementing them. In general, developing the UI to display the data and provide user interaction requires considerably more effort.

Qualitatively, implementing vizketches never required thinking about distributed systems or concurrency. A developer simply provides the *summarize* and *merge* functions, which are purely local, while the rest of Hillview takes care of the distributed systems aspects of running vizketches across many cores and servers. Of course, we had to implement the distributed execution framework for vizketches in Hillview, but this implementation was done once and benefits all vizketches, including future extensions.

7.5 Hillview effectiveness: case study

We next consider the question of how effective Hillview is to browse and answer queries on large datasets. We address this question through a case study.

Workload. A person who is not familiar with Hillview examines the Flights-1x data set and formulates a set of questions (shown in Figure 10) that interests her and that she thinks the dataset answers.

Setup. The experiment is carried out by an operator who is familiar with Hillview well but does not know the questions ahead of time. In each experiment, we show a question to the operator and ask him to answer it using Hillview. Our goal is to understand if the spreadsheet is powerful enough to answer the question and, if so, how easily it can do that. Note that this experiment does not evaluate ease-of-use by beginners, because the operator is an expert. This is intentional: Hillview users are not casual users but data analysts, whose job is dedicated to explore data and so they can obtain the required training.

Question	Actions	Time	Question	Actions	Time
Q1	5	1:11	Q11	3	1:18
Q2	3	1:32	Q12	5	6:44
Q3	4	1:13	Q13	6	6:27
Q4	5	0:47*	Q14	2	0:20
Q5	5	2:26	Q15	4	1:56
Q6	4	2:15*	Q16	3	1:07
Q7	2	1:08	Q17	3	1:07
Q8	5	2:56	Q18	2	1:08
Q9	1	0:34	Q19	2	0:40
Q10	1	1:08*	Q20	—	2:23†

Figure 11: Number of actions and time in minutes:seconds required for an operator to answer questions using Hillview. Most of the time is spent thinking about how to best translate a question into a set of UI operations. Notes: *These queries had only a partially satisfactory answer. †In this question, the data set did not have enough information to answer it; the measured time is how long it took to make that determination.

For each question, we measure the time and number of spreadsheet actions that the operator takes to answer the question. A spreadsheet action consists of choosing an operation on a menu, clicking on the spreadsheet, or dragging the mouse to select a region. For example Q1 can be answered by filtering the main table for column Airline=UA, producing a histogram on DepartureDelay, then going back to the main table and filtering for column Airline=AA, producing a second histogram on DepartureDelay. To answer the question, we hover the mouse over the histograms to find the delay percentiles.

Results. Figure 11 shows the results. Answering a question took at most 6:44 (minutes:seconds), with most questions taking less than 2:30 (all except three). The average and median times are 1:57 and 1:12. Most of the time is the operator thinking about what to do, rather than waiting for the spreadsheet to respond (if the operator knew exactly what to do, all queries could be answered in under 30 seconds). The minimum and maximum number of actions were 1 and 6, with mean and median 3.4 and 3. Queries Q4, Q6 and Q10 did not have completely satisfactory answers because the spreadsheet cannot clearly separate dates (Q4, Q10) or the spreadsheet did not merge and deduplicate the destinations (Q6). Question Q20 could not be answered because the dataset does not have the information (e.g., we discovered that it lacks the downed flights on 9/11). We see that Hillview was effective at addressing most queries at small times, showing that (1) Hillview implements enough functionality to be usable and (2) it provides a interactive experience for human timescales.

8. RELATED WORK

Hillview is the first spreadsheet to scale massively with interactive speed. Hillview borrows ideas from the algorithms and computer graphics literature, namely mergeable summaries [2] (or sketches) and visualization-driven computation; it uses relies on many techniques from databases (approximate query processing, on-line analytics), big-data analytics, and distributed systems.

Hillview follows Shneiderman's visualization mantra [85]: "overview first, zoom and filter, details on demand". Fisher [38] identifies principles for interactively visualizing big data ("look at less of it" and "look at it faster"); these principles guided the design of vizketches.

Big data visualization is a broad area; we give an overview of the closest related work below. For more information, we refer the reader to several surveys in the area [82, 42, 41, 8, 9]. Compared to published systems, Hillview achieves the best scalability for the amount of resources: we are not aware of any system that can handle a trillion cells with only 8 servers.

Distributed visualization engines. Hillview evolved from Sketch [14], which proposes a distributed data exploration library with applications to a performance analyzer and a spreadsheet. VisReduce [50] provides incrementally updated approximations of visualizations computed over progressively larger samples. Vizdom [23] is a simple UI for data manipulation and exploration; it runs on top of the A-WARE smart caching and streaming engine [24] and uses the Tuppleware analytics system [22].

Visualization using big data query engines. One way to visualize big data is to connect a visualization engine to an analytics engine, such as Hive [93], Impala [60], Presto [78], Dremel [65] (commercialized as BigQuery), Drill [46], PowerDrill [45], Spark [103], Druid [102], or Pinot [49]. This approach has advantages: it reduces design effort by using existing systems, and it leverages the years of effort spent in their optimization. However, this approach does not achieve the speed needed for a spreadsheet: the generality of analytics engine imposes overheads and computes unnecessary results, since there is no integration with the visualization engine. Several systems follow this approach. Microsoft PowerBI [67] using DirectQuery [26] and Polaris/Tableau [90, 98, 99] provide plug-ins to many analytics engines; as discussed in [17], the users of such systems have to carefully avoid many queries that cannot be answered efficiently. IBM BigSheets [12] computes interactively only over a subset of the data; once the user settles on a query, it is actually run in batch mode using Spark. HadoopVis [32] uses Hadoop to render geo-spatial data. ScalaR [6] uses relational databases; the system in [96] uses MapReduce for mesh rendering and isosurface extraction. SwiftTuna [51, 52] uses Spark. OmniSci [72] uses GPUs in one machine for server-side rendering.

Facebook's Scuba [1] has been used as a back-end to visualization systems. Scuba provides fast response times but with a different trade-off between data scale, responsiveness, and correctness. Scuba computes much more than "what you can see", since its compute engine is decoupled from its visualization. Thus, queries might return unbounded amounts of data to the visualization engine, hampering real-time responsiveness. To avoid that, Scuba truncates worker responses to 100,000 rows ([1, page 4]) and omits workers that do not respond in 10ms ([1, page 6]). This can produce arbitrarily incorrect visualizations.

The vizketch computational model is similar to the MPI Reduce [87] primitive used in supercomputing, to the Neptune system [18], to the architecture of log analytics systems such as Splunk [88], and to aggregation networks for sensor networks [79]; these are general-purpose platforms, and not visualization systems.

Sampling and indexing. Sampling and indexing are used to accelerate visualization in many systems. [15] considers the problem of sampling a database for minimizing the error for a given set of queries. BlinkDB [3] uses stratified sampling, which is effective, but leaves the burden on users to write appropriate SQL queries and find appropriate error and time bounds. Smart sampling is used by [39]. [97] uses stratified sampling to accelerate queries in log management systems. [101] uses stratified sampling in Scope to reduce sample sizes while minimizing errors; samples are incrementally maintained. Pangloss [69] uses "optimistic" visualization on sampled data to provide fast results.

The idea of using perceptual limitations to drive sampling appears first in [30]. [58] uses perceptual limitations and sampling algorithms for specific chart types (e.g., bar charts). M4 [53] uses the screen resolution to rewrite SQL queries to compute reduced results suitable for renderings of line plots; this is extended for other chart types in VDDA [54]. Sample+Seek [29] executes responsive aggregated queries on a single table; it uses measure-biased sampling together with new indexing schemes, specific to the aggregation computed, to minimize errors. G-OLA [104] handles interactive aggregate OLAP queries over massive data sets.

VAS [76] samples data to minimize the visualization errors for scatter-plots. SynopViz [10] and Skydive [43] build hierarchical multi-scale models of the data for browsing linked data sets.

Progressive analytics. Hillview visualizations are incrementally updated; this technique is called online aggregation [47] or progressive analytics [39, 89, 94, 35]. There is significant work on this topic. MapReduce Online [20, 75] is based on MapReduce. EARL [61] uses statistical boostrapping for providing reliable online early estimates for the output of MapReduce computations. Progressive Insights [89] finds common subsequences in event series of medical records, focusing on its UI design for incremental display and exploration. PIVE [16] adapts computation to limited screen resolution for iterative algorithms (such as clustering or dimensionality reduction). DimXplorer[94] performs progressive computation and rendering of dimensionality reduction operations (such as clustering and PCA); it uses sampling for fast response times. Stat! [5] operates in conjunction with a streaming engine, and presents immediately incremental results. Microsoft's Tempe [66] runs on top of a streaming engine and provides progressive visualizations.

All above systems lack some of the benefits of vizketches: parallelization, computation efficiency, and bandwidth efficiency (§4.4), which are required for Hillview.

Nanocubes [62], imMens [63], and Hashedcubes [74] improve interactivity by pre-processing the data to build smart indexes. DICE [55], Sesame [56] and ForeCache [7] use sessions and locality to pre-compute views or to reuse computation results between consecutive views. These ideas improve interactivity, but restrict the scope of queries to pre-processed columns (e.g., the user preselects a few columns to optimize) or spend significant time for pre-processing. By contrast, Hillview uses no pre-processing or indexes, because we do not know ahead of time which columns the user might choose to explore.

VisTrees are indexes designed to support quick histogram construction for visualizations [31]. Profiler [57] and Foresight [28] propose methods to find abnormality in the data; Hillview could incorporate this functionality, especially Foresight which is based on sketches. NeedleTail [59] uses a small in-memory index to allow fast browsing and displaying any-k records. AQP++ [77] combines approximate query processing with aggregate pre-computation.

9. CONCLUSION

Hillview is a spreadsheet that supports a trillion cells even with a modest number of servers. Hillview introduces a new query execution engine specialized to render tabular views and charts for a spreadsheet. The new engine uses vizketches, a new but simple idea that parallelizes computation and calculates only what is needed for a good visualization. We believe Hillview is a useful tool for humans to explore data; it nicely complements other tools, such as analytics frameworks, which have other uses.

10. REFERENCES

[1] L. Abraham, J. Allen, O. Barykin, V. R. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at Facebook. *PVLDB*, 6(11):1057–1067, 2013.

[2] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *ACM SIGMOD International conference on Management of data*, pages 23–34, 2012.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[4] Apache Tomcat. http://tomcat.apache.org. Retrieved March 2019.

[5] M. Barnett, B. Chandramouli, R. DeLine, S. Drucker, D. Fisher, J. Goldstein, P. Morrison, and J. Platt. Stat!: an interactive analytics environment for big data. In *ACM SIGMOD International conference on Management of data*, pages 1013–1016, 2013.

[6] L. Battle, R. Chang, and M. Stonebraker. Dynamic reduction of query result sets for interactive visualization. In *IEEE International Conference on Big Data*, pages 1–8, Oct 2013.

[7] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *International Conference on Management of Data (SIGMOD '16)*, pages 1363–1375, 2016.

[8] M. Behrisch, D. Streeb, F. Stoffel, D. Seebacher, B. Matejek, S. H. Weber, S. Mittelstaedt, H. Pfister, and D. Keim. Commercial visual analytics systems – advances in the big data analytics field. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[9] N. Bikakis. Big data visualization tools. In S. Sakr and A. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–6. Springer International Publishing, Cham, 2018.

[10] N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis. A hierarchical aggregation framework for efficient multilevel visual exploration and analysis. *Semantic Web*, 8(1):139–179, 2017.

[11] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization and Comp. Graphics (Proc. InfoVis)*, 2011.

[12] M. Brown. BigSheets for the common man. https://www.ibm.com/developerworks/library/bd-bigsheets/index.html, December 2013.

[13] M. Budiu, P. Gopalan, L. Suresh, U. Wieder, H. Kruiger, and M. K. Aguilera. Hillview: A trillion-cell spreadsheet for big data (extended version). http://github.com/vmware/hillview/tree/master/docs/paper.pdf, 2019.

[14] M. Budiu, R. Isaacs, D. Murray, G. Plotkin, P. Barham, S. Al-Kiswany, Y. Boshmaf, Q. Luo, and A. Andoni. Interacting with large distributed datasets using Sketch. In *Eurographics Symposium on Parallel Graphics and Visualization*, Groningen, Netherlands, June 6-7 2016.

[15] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM SIGMOD International conference on Management of data*, pages 295–306, 2001.

[16] J. Choo, C. Lee, H. Kim, H. Lee, C. Reddy, B. Drake, and H. Park. PIVE: Per-iteration visualization environment for supporting real-time interactions with computational methods. In *Visual Analytics Science and Technology (VAST)*, 2014.

[17] R. Christopher and V. Krishnan. Optimizing your Amazon Redshift and Tableau software deployment for better performance v2. https://www.tableau.com/sites/default/files/whitepapers/optimizing_tableau_aws_redshift__whitepaper__v2.pdf, 2017.

[18] L. Chu, H. Tang, T. Yang, and K. Shen. Optimizing data aggregation for cluster-based Internet services. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 119–130, 2003.

[19] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 225–234, New York, NY, USA, 2007. ACM.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[21] G. Cormode. Data sketching. *Communications of the ACM*, 60(9):48–55, Aug. 2017.

[22] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, Aug. 2015.

[23] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2027, Aug. 2015.

[24] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The case for interactive data exploration accelerators (IDEAs). In *Human-In-the-Loop Data Analytics (HILDA)*, pages 11:1–11:6, 2016.

[25] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, J. Watt, J. Ferraiolo, F. Jun, and D. Jackson. Scalable vector graphics (SVG) 1.1. https://www.w3.org/TR/SVG/, August 2011.

[26] K. de Jonge. DirectQuery in SQL server 2016 analysis services. http://download.microsoft.com/download/F/6/F/F6FBC1FC-F956-49A1-80CD-2941C3B6E417/DirectQuery%20in%20Analysis%20Services%20-%20Whitepaper.pdf, January 2017.

[27] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[28] Ç. Demiralp, P. J. Haas, S. Parthasarathy, and T. Pedapati. Foresight: Recommending visual insights. *PVLDB*, 10(12):1937–1940, 2017.

[29] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In *ACM SIGMOD International conference on Management of data*, pages 679–694, 2016.

[30] A. Dix and G. Ellis. *by chance:* enhancing interaction with large data sets through statistical sampling. In *Advanced Visual Interfaces*, pages 167–176, 2002.

[31] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. VisTrees: fast indexes for interactive data exploration. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*, page 5, 2016.

[32] A. Eldawy, M. F. Mokbel, and C. Jonathan. HadoopViz: A MapReduce framework for extensible visualization of big spatial data. In *International Conference on Data Engineering (ICDE)*, Helsinki, Finland, May 2016.

[33] N. Elmqvist and J. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, May 2010.

[34] :::fastutil: Fast and compact type-specific collections for Java. http://fastutil.di.unimi.it. Retrieved October 2017.

[35] J.-D. Fekete and R. Primet. Progressive analytics: A computation paradigm for exploratory data analysis. https://arxiv.org/abs/1607.05162, 2016.

[36] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributing symmetric streaming computations. *ACM Trans. Algorithms*, 6(4):66:1–66:19, 2010.

[37] I. Fette and A. Melnikov. The WebSocket protocol. IETF RFC 6455, December 2001.

[38] D. Fisher. Big data exploration requires collaboration between visualization and data infrastructures. In *Human-In-the-Loop Data Analytics (HILDA)*, pages 16:1–16:5, 2016.

[39] D. Fisher, I. Popov, S. Drucker, and M. Schraefel. Trust me, I'm partially right: Incremental visualization lets analysts explore large datasets faster. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 1673–1682, 2012.

[40] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms (AofA) DMTCS proc.*, pages 127–146, 2007.

[41] A. Ghosh, M. Nashaat, J. Miller, S. Quader, and C. Marston. A comprehensive review of tools for exploratory analysis of tabular industrial datasets. *Visual Informatics*, 2018.

[42] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2142–2157, 2016.

[43] P. Godfrey, J. Gryz, P. Lasek, and N. Razavi. Visualization through inductive aggregation. In *International Conference on Extending Database Technology (EDBT)*, pages 600–603, 2016.

[44] gRPC: A high performance, open-source universal RPC framework. https://grpc.io/. Retrieved October 2017.

[45] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, July 2012.

[46] M. Hausenblas and J. Nadeau. Apache Drill: Interactive ad-hoc analysis at scale. *IEEE Comput. Graph. Appl.*, 1(2), June 2013.

[47] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM SIGMOD International conference on Management of data*, pages 171–182, 1997.

[48] J. F. Hughes, A. van Dam, M. McGuide, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley. *Computer Graphics: Principles and Practice (3rd Edition)*. Addison-Wesley Professional, 2013.

[49] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and R. Aringunram. Pinot: Realtime OLAP for 530 million users. In *International Conference on Management of Data (SIGMOD)*, pages 583–594, 2018.

[50] J.-F. Im, F. G. Villegas, and M. J. McGuffin. VisReduce: Fast and responsive incremental information visualization of large datasets. In *IEEE International Conference on Big Data*, pages 25–32, Oct 2013.

[51] J. Jo, W. Kim, S. Yoo, B. Kim, and J. Seo. SwiftTuna: Incrementally exploring large-scale multidimensional data. In *IEEE VIS*, Phoenix, AZ, October 2016.

[52] J. Jo, W. Kim, S. Yoo, B. Kim, and J. Seo. SwiftTuna: Responsive and incremental visual exploration of large-scale multidimensional data. In *Pacific Visualization Symposium (PacificVis)*, pages 131–140, Seoul, Korea, 2017.

[53] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *PVLDB*, 7(10):797–808, June 2014.

[54] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. VDDA: Automatic visualization-driven data aggregation in relational databases. *The VLDB Journal*, 25(1):53–77, Feb. 2016.

[55] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed interactive cube exploration. In *International Conference on Data Engineering (ICDE)*, pages 472–483, March 2014.

[56] N. Kamat and A. Nandi. A session-based approach to fast-but-approximate interactive data cube exploration. *ACM Trans. Knowl. Discov. Data*, 12(1):1–26, Feb. 2018.

[57] S. Kandel, R. Parikh, A. Paepcke, J. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Advanced Visual Interfaces*, 2012.

[58] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *PVLDB*, 8(5):521–532, Jan. 2015.

[59] A. Kim, L. Xu, T. Siddiqui, S. Huang, S. Madden, and A. Parameswaran. Optimally leveraging density and locality for exploratory browsing and sampling. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA 18)*, HILDA, pages 7:1–7:7, 2018.

[60] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *Conference on Innovative Data Systems Research (CIDR '15)*, January 4-7 2015.

[61] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on MapReduce. *PVLDB*, 5(10):1028–1039, June 2012.

[62] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[63] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[64] E. Meijer. Your mouse is a database. *ACM Queue*, 10(3):20–33, Mar. 2012.

[65] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel:

Interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, Sept. 2010.
[66] Microsoft Corp. Tempe. http://research.microsoft.com/en-us/projects/tempe/. Retrieved January 2019.
[67] Microsoft PowerBI. https://powerbi.microsoft.com. Accessed October 2017.
[68] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.
[69] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *ACM Human Factors in Computing Systems (CHI)*, 2017.
[70] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and trends in theoretical computer science. Now Publishers, 2005.
[71] U. D. of Transportation. Airline on-time performance data. https://transtats.bts.gov/Tables.asp?DB_ID=120. Retrieved January 2019.
[72] OmniSci is the extreme analytics platform. https://www.omnisci.com, Retrieved October 2018.
[73] Oracle Corp. Project Nashorn. http://openjdk.java.net/projects/nashorn/. Retrieved February 2018.
[74] C. A. L. Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):671–680, 2017.
[75] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large MapReduce jobs. In *PVLDB*, Seattle, WA, August 2011.
[76] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *International Conference on Data Engineering (ICDE)*, pages 755–766. IEEE, 2016.
[77] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: Connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 1477–1492, 2018.
[78] Presto: Distributed SQL query engine for big data. https://prestodb.io/, Retrieved 2018.
[79] R. Rajagopalan and P. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8(4):48–63, 2006.
[80] ReactiveX: An API for asynchronous programming with observable streams. http://reactivex.io/. Retrived October 2017.
[81] R. Rubinfeld and A. Shapira. Sublinear time algorithms. *SIAM J. Discret. Math.*, 25(4):1562–1588, Nov. 2011.
[82] C. Scheidegger. Interactive visual analysis of big data. In P. Bühlmann, P. Drineas, M. Kane, and M. van der Laan, editors, *Handbook of Big Data*. Taylor and Francis group, 2016.
[83] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014.
[84] J. Shlens. A tutorial on principal component analysis. https://arxiv.org/abs/1404.1100, 2014.
[85] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343, Boulder, CO, September 1996.
[86] B. Shneiderman. Extreme visualization: squeezing a billion records into a million pixels. In *ACM SIGMOD international conference on Management of data (SIGMOD 2008)*, pages 3–12. ACM, 2008.
[87] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.
[88] Splunk: Dashboards and visualizations. http://docs.splunk.com/Documentation/Splunk/latest/Viz/Aboutthismanual.
[89] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, December 2014.
[90] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.
[91] M. Stonebraker, S. Madden, D. J. Abadi, S. avros Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *International Conference of Very Large Data Bases (VLDB)*, pages 1150–1160, Sept. 2007.
[92] M. Thorup. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 371–380, New York, NY, USA, 2013. ACM.
[93] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, Aug. 2009.
[94] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser. Designing progressive and interactive analytics processes for high-dimensional data analysis. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):131–140, Jan. 2017.
[95] TypeScript: JavaScript that scales. http://www.typescriptlang.org/. Retrieved October 2017.
[96] H. Vo, J. Bronson, B. Summa, J. Comba, J. Freire, B. Howe, V. Pascucci, and C. Silva. Parallel visualization on large clusters using MapReduce. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 81–88, oct. 2011.
[97] T. Wagner, E. Schkufza, and U. Wieder. A sampling-based approach to accelerating queries in log management systems. In *International Conference on Systems, Programming, Languages and Applications: Software for Humanity, (SPLASH)*, Amsterdam, Netherlands, October 2016.
[98] R. Wesley, M. Eldridge, and P. T. Terlecki. An analytic data engine for visualization in Tableau. In *ACM SIGMOD International conference on Management of data*, pages 1185–1194, 2011.
[99] R. M. G. Wesley and P. Terlecki. Leveraging compression in the Tableau data engine. In *ACM SIGMOD International conference on Management of data*, pages 563–573, 2014.
[100] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems: Vision paper. *PVLDB*, 7(10):903–906, June 2014.
[101] Y. Yan, L. J. Chen, and Z. Zhang. Error-bounded sampling

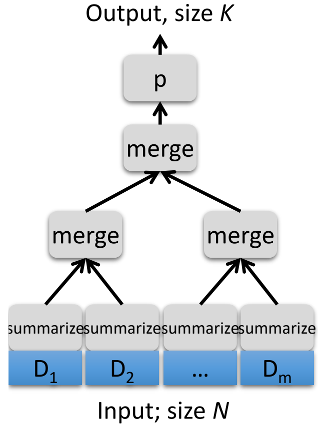


Figure 12: Abstract computational model for vizketches.

for analytics on big sparse data. In *International Conference of Very Large Data Bases (VLDB)*, Hangzhou, China, September 1-5 2014.

[102] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *ACM SIGMOD International conference on Management of data*, pages 157–168, 2014.

[103] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.

[104] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *ACM SIGMOD International conference on Management of data*, pages 913–918, 2015.

APPENDIX

A. A COMPUTATIONAL MODEL FOR VIZKETCHES

The abstract computational model for vizketches is illustrated in Figure 12: consider a rooted tree T with m leaves[7] $\{\ell_1,\ldots,\ell_m\}$. Each leaf is a server that ℓ_i contains a local dataset D_i, which is usually a multiset of values from a domain $\mathscr{D}$. The various ℓ_is can run computations in parallel. Let $D = \uplus_{i=1}^{n} D_i$ denote the entire input which is also a multiset over $\mathscr{D}$ (where $\uplus$ denotes disjoint multiset union). We will denote with $N = |D|$ the size of the input data. These same servers can also servers as internal nodes (and root) of the tree.

Let $\mathscr{V}$ be the space of data views (histograms, for instance). There is a client who does not know the input, and who wishes to compute a function $f(\mathscr{D}) \in \mathscr{V}$, where each element of $\mathscr{V}$ is v bits long, and is essentially *independent of the size of the input* N. We stress that f is a function on multisets from $\mathscr{D}$, and not on sequences, so it is oblivious to how the records are partitioned among the servers and how each server orders them.

A computation protocol proceeds in k rounds for some integer k, in all the vizketches we implement k was 1 or 2. In round j where $j \in [k]$, the root issues a request r^j, which is broadcast to all servers. The request is based on the function f and the results of previous rounds. The computation then proceeds in two phases:

- **Summarize:** Every leaf makes a pass over their respective input and computes a function $summarize^j : D_i \to \mathscr{V}$ and sends it to its parent in T. While each round could do something different the description length of elements output is bounded by v for every round j.
- **Merge:** This phase applies an aggregation function $merge^j : \mathscr{V}^d \to \mathscr{V}$ at each internal node where d is the number of children. The function aggregates results from the children and sends the aggregated results to the parent of v in T. The inputs to *merge* are from the same small domain $\mathscr{V}$, so the computational cost of *merge* is dominated by the cost of the creation function *summarize*.

At the end of the last round, the root computes the output $\hat{f}(D)$. Protocols may be randomized. Approximation is often essential since computing even simple functions exactly can be hard in this model. There are two requirements from the protocol:

- **Correctness:** $\hat{f}(D)$ approximates $f(D)$, under a suitably defined notion of approximation (with high probability for a randomized protocol).
- **Efficiency:** The length of all messages communicated in the protocol and the memory needed to compute them are $\mathrm{polylog}(v, \log(n))$.

This model is closely related to well-studied models in the fields of algorithms and databases. The *summarize* phase relies on sampling and streaming algorithms [81, 70]. The *merge* phase and the model for incremental computations is closely related to the sketching model and massive unordered data (MUD) model [21, 36] of distributed computation and the notion of mergeable summaries [2]. At the same time, the combination of the restriction on the size of outputs in $\mathscr{V}$ and the efficiency requirement for *summarize*are specific to visualization. This implies that our computational model is *less* general than map-reduce [27] where the communication between machines can have messages of unbounded size.

[7]The tree does not have to be balanced as in this figure.

B. VIZKETCH ALGORITHMS

We present details for all classes of vizketches used in Hillview.

B.1 Vizketches for charts

We describe the vizketches used in Hillview for producing various charts. Here, a vizketch is parameterized by the target display resolution, and produces calculations that are just precise enough to render at that resolution. We now give the details for each vizketch. In appendix C, we give rigorous guarantees for correctness of the underlying algorithms

Equi-width buckets for string data. Hillview can plot charts for arbitrary string data. It divides the data range into equi-width bins. If there are few distinct values (50 or fewer), we assign a bin for each value. Otherwise, we consider an alphabetical ordering of strings and aggregate contiguous values into bins, so that the number of bins does not exceed 50. The challenge is to find bin boundaries without sorting the full dataset. This is done using a sketch based on bottom-k sampling [92, 19], which is an efficient mergeable randomized streaming algorithm that computes *approximate quantiles* over *distinct* strings, where the quantiles are set to 1/50, 2/50, etc.

Cumulative distribution functions (CDFs). We are given a column with numeric values in a range $[x_0, x_1)$, and a target screen dimension of $H \times V$. Horizontally, a pixel h represents an interval $I_h = [x_0, x_0+(h+1)(x_1-x_0)/H)$. We must determine how many data points belongs to I_h and divide by the total number of points. Even if we compute the cdf exactly, when we render it on the

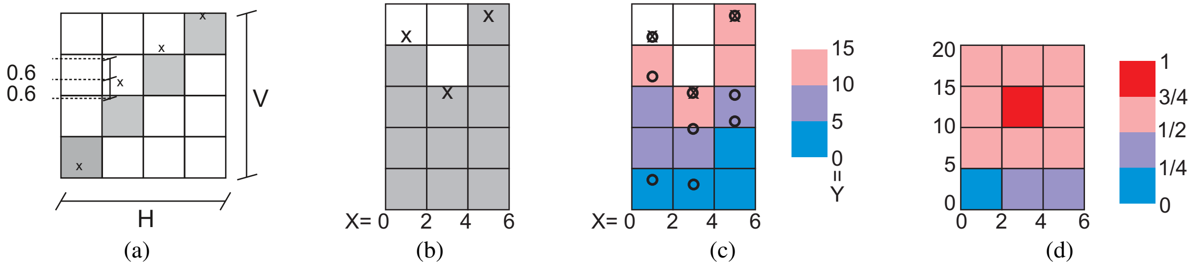


Figure 13: Charts in Hillview have an error of at most one pixel or one color shade with high probability. (a) A cdf plot with dimension $H \times V$ pixels. The $\times$ in the figure indicates the exact value of the cdf. The grey box indicates the rendering, which is at most 0.5 pixel away from the $\times$. (b) a histogram with three bars. The $\times$ indicates the correct height for the bar—again, at most one pixel away from the rendering. (c) A stacked histogram with three bars. The bars show a histogram for the first variable X. Each bar has three subdivisions, each with a different color, which indicates the value of the second variable Y. The size of the subdivision shows the density for the bin with both variables. The $\circ$ indicates the correct location for the subdivision. (d) A heat map. The x-axis has bins for the first variable; the y-axis, for the second variable. The color indicates the density of each bin, where the error is at most one color shade with high probability.

screen, we round it to the nearest pixel, which introduces a *quantization error* of $\pm 0.5/V$. A simple way to compute the cdf exactly is to scan the dataset and count, however, it requires a full scan and many counters. In our renderings, our goal is to compute, for each horizontal pixel of the cdf, a value between 0 and 1 that lies within $\pm 0.1/V$ of the true cdf value, so that after rounding, we render a pixel value that is at most $\pm 0.6/V$ away from the true cdf value (and the pixel itself is at most one away for the pixel that would be rendered if we did an exact computation (Figure 13(a)). The advantage now is that we can approximate the cdf to the desired accuracy level with probability $1-\delta$ by just sampling the data. Our calculation for the required sample size are based on a standard Chernoff bound for the error per pixel. We then use the fact that pixels are arranged on a line. In [13] Appendix C we show the target sample size is $n = O(V^2 \log(1/\delta))$. The *summarize* function thus samples the dataset with the appropriate rate and counts the number of values that fall in each interval I_h. The function outputs a small vector S of H bin counts. The *merge* function of the vizketch takes two such vectors and adds them. To produce the cdf visualization from the vizketch, we scale each bin of S by V/n to obtain a value in $0 \ldots V-1$.

CDFs for string data. We can produce CDF plots even for string data, by combining the equi-width bucket computation with the counting-based CDF computation described in the previous paragraph.

Histograms. We are given a numerical column (or a value that can be readily converted to a real number, such as a date) with range $[x_0, x_1)$, a number B of histogram bars, and their maximum pixel height V. The histogram vizketch (Figure 13(b)) is similar to the cdf vizketch, except that it has B bins instead of H horizontal pixels and we now divide the range $[x_0, x_1)$ into H equi-sized intervals, one per bin. To maximize use of screen, we should scale the bars so that the largest one has V pixels. We show in Appendix C of [13] that to bound the error of a bar to one pixel with probability $1-\delta$, requires sampling a set of size $n = O(V^2 B^2 \log(1/\delta))$. The *summarize* function outputs a vector of B bin counts, and the *merge* function adds two vectors.

Histograms for text data. We allow users to plot histograms for arbitrary text data; the number of bars is limited to 50, and if the number of distinct strings is larger each bar represents a set of strings that are contiguous in alphabetical order. The user can zoom-in to reveal the finer-grained structure of each bar. These are based on the equi-width buckets computation.

Stacked histogram. A stacked histogram (Figure 13(c)) involves two columns X and Y, which can be numeric or categorical. We are given two input ranges $[x_0, x_1)$ and $[y_0, y_1)$, a number B_x of histogram bars for X, a number B_y of colors for bins for Y, and pixel dimensions $H \times V$. The human eye cannot distinguish many colors reliably, so B_y is limited to $\approx$20. The stacked histogram represents counts in two ways: (1) the height of each histogram bar represents counts of bins of X (like a histogram), (2) the height of a subdivision of a bar represents counts of a bin of Y within the bin of X of that bar. To bound the error by a pixel, the accuracy of (1) should be ± 0.6. As for (2), in the worst case a subdivision will be the entire bar, which also requires an accuracy of ± 0.6. The required target sample size is $n = O(V^2 B_x^2 \log(1/\delta))$. The *summarize* function thus samples the dataset with the required rate, counting the number of values that fall in each X bin, as well as the number of values that fall in the combined X, Y bins. The function outputs a small vector S of $B_x + B_x \times B_y$ bin counts. The *merge* function of the vizketch takes two such vectors and adds them.

Histogram (streaming). We also provide a simple vizketch to compute histograms without sampling, if users want to get the results precise to the last digit. Given a numerical column with range $[x_0, x_1)$, a number B of histogram bars, the *summarize* function scans the data and counts the number of items in each bin, producing a vector. The *merge* function adds two vectors.

Normalized stacked histogram. This visualization is like a stack histogram except that the size of each bar is normalized to 1. This difference, however, impacts the required accuracy for subdivisions; for example, if a bin of X has a small count, it gets normalized to a full bar and therefore its subdivisions require a higher accuracy. To implement this visualization, we use a stacked histogram sketch without sampling.

Heat map. We are given two columns X and Y with ranges $[x_0, x_1)$ and $[y_0, y_1)$, and the pixel dimensions $H \times V$. A heat map (Figure 13(d)) defines bins in two dimensions, where each bin consumes $b\times$ pixels, where $b = 3$. Thus, we have $B_x = H/b$ and $B_y = V/b$ bins for X and Y. The density of a bin is represented by a color scale. If we use $c\approx$20 distinct colors, the required accuracy for each bin density is $1/2c$. This requires a target sample size $n = O(c^2 B_x^2 B_y^2 \log(1/\delta))$ (sampling can only be used if the map-

ping from count to color is linear, otherwise the full dataset has to be scanned). The *summarize* function samples data with the target rate, counting the number of values that fall in each bin. It outputs a matrix of $B_x \times B_y$ bin counts. The *merge* function adds two such matrices.

Trellis plots. A Trellis plot displays a 1D or 2D array of other plots; in Figure 13 we show a Trellis plots of heatmaps. For a 1D Trellis plot are given three columns W, X, and Y; k elements $w_1, \ldots, w_k$ of W; the ranges $[x_0, x_1)$ and $[y_0, y_1)$ of X and Y; and pixel dimensions $H \times V$. A heat map trellis plot produces k heat maps, each for a fixed range of values w_i in column W. This might appear like significant computation, but because the rendering area is limited to $H \times V$, a large number of heat maps means that each heat map is small. For example, if we render the k heat maps as a $2 \times k/2$ matrix of heat maps, then each heat map has dimension $H/2 \times 2V/k$. The vizketch computes all heat maps in parallel, but due to the quadratic dependency on the number of bins, this requires a smaller sample size than rendering a single heat map of the same pixel dimensions. The *summarize* function outputs the same number of bins as a single heat map of the same pixel dimensions.

B.2 Vizketches for tabular view

The next several vizketches are used to produce the tabular views of the spreadsheet. They follow the principle of calculating just what needs to be displayed, with approximations where possible.

Next items. This vizketch is used to render a view of the spreadsheet given the current top row R (or $R = \perp$ to choose the beginning). We are also given a column sort order, and the number K of rows to show. This vizketch returns the contents of the K rows that follow R in the sort order. The *summarize* function scans the dataset and keeps a priority heap with the K next values following row R in the sort order. The *merge* function combines the two priority heaps by selecting the smallest K elements and dropping the rest.

Quantile for scroll bar. When a user moves the scroll bar she indicates a quantile of the data to be displayed. The tabular will start display the appropriate quantile of the current sorting order corresponding to the scroll bar position. For example, if the scroll bar is in the middle, we move to the median of the sorting column. Given a scroll with V pixels and a position v in $0 \ldots V-1$, this vizketch uses a quantile sketch method with target quantile v/V and accuracy ± 0.6, which samples $O(V^2)$ random rows and returns the quantile from this set. The method returns the contents of a row that becomes the top entry in the tabular view.

Find text. This vizketch implements the free-form text find functionality of the spreadsheet. Given a row R, a search criteria (the search text; whether it is exact match, substring, or regexp; and whether it is case sensitive), and a column sort order, we want to find the next row satisfying the criteria in the sort order. This is similar to the next item vizketch above except that we eliminate all rows that do not match the search criteria.

Heavy hitters (streaming). This vizketch serves to find the most frequent elements of a column and their count. More precisely, given a column and a maximum number K of items, we want to find elements that occur more than a fraction $1/K$ of the time. We also want the approximate counts for such elements. To do this, we directly use the Misra-Gries streaming algorithm [68].

Heavy hitters (sampling). Another vizketch to find heavy hitters works by sampling. Given a probability of error δ and let K be as above. The basic idea is to sample with a target size n (determined below), and select an item as a heavy hitter if it occurs with frequency at least $3n/4K$. A statistical calculation shows that by picking $n = K^2 \log(K/\delta)$, with probability $1-\delta$ we can obtain all elements that occur more than $1/K$ of the time and no elements that occur fewer than $1/4K$ of the time. This method is particularly efficient if K is small. In fact, our experiments indicate this method is better the previous one when $K \geq 1/100$.

Spreadsheet functionality	**Required vizketch(es)**
Drawing any chart	Range + chart-specific vizketch
Initial tabular view	Next items
Scroll up/down table	Next items
Moving scrollbar	Quantile + next items
Find text	Filter + next items
Heavy hitters	Heavy hitters
PCA	PCA

Figure 14: Using vizketches to implement specific spreadsheet functionalities.

B.3 Vizketches for auxiliary functionality

Hillview uses vizketches to obtain general information about columns. Technically, these vizketches are just sketches because they need not be tuned for a target visualization. We list them here because in Hillview they are executed by the same mechanism as the other vizketches.

Moments. Given a column, this vizketch collects its minimum and maximum values, number of rows, the number of missing values, and the statistical moments up to a specified value K (including mean and variance, the first two moments). This information is used to select the range of certain visualizations (histograms, heat maps, etc). The information is also shown to the user if she requests a summary of the column data.

Number of distinct elements. This information is computed approximatively using the HyperLogLog sketch [40].

Principal component analysis. PCA can summarize M numeric columns into $K<M$ columns, by projecting the $M \times N$ matrix into a $K \times N$ matrix along the eigen vectors of the $M \times M$ correlation matrix. This matrix can be efficiently computed by a sampling-based sketch.

B.4 Vizketches in the spreadsheet

Spreadsheet actions are implemented using one or more vizketches (Figure 14). All charts, when produced initially, require a vizketch to determine the range of the inputs; subsequently, this information can be cached. Changing the tabular view requires the next items vizketch to produce the new visible rows sorted the chosen order. The scrollbar is implemented by a quantile computation (e.g., scrollbar in the middle corresponds to the median), following by next items to produce the new visible rows. Specific analyses run the corresponding vizketches (e.g., Heavy hitters).

C. PROOFS OF CORRECTNESS

Consider a distribution $\mathscr{P}$ on some domain $\mathscr{D}$ which might be discrete or continuous. Let $\mathscr{F}$ be a family of subsets of $\mathscr{D}$. For $I \in \mathscr{F}$, let $\mu_{\mathscr{P}}(I)$ be its measure under $\mathscr{P}$. We draw a sample S of n of size n from $\mathscr{D}$ according to $\mathscr{P}$. We expect $|S \cap I|/|S|$, the fraction of points in S that land in S, to be close to $\mu_{\mathscr{P}}(I)$ for every set I. The VC dimension theorem relates the sample size needed for this to the VC-dimension (see [83, Chapter 6]) of the family of subsets.

THEOREM 1. *[83, Theorem 6.8] Let $\mathscr{P}$ be a distribution on $\mathscr{D}$, and let $\mathscr{F} \subseteq 2^{\mathscr{D}}$ be a family of subsets with VC-dimension d. Let S be a (multi)-set of samples drawn from $\mathscr{P}$ where $|S| \geq C(d + \log(1/\delta))/\varepsilon^2$. Then with probability $1-\delta$, for every set $I \in \mathscr{F}$, we have*

$$\left| \frac{|S \cap I|}{|S|} - \mu_{\mathscr{P}}(I) \right| \leq \delta. \tag{1}$$

We will use this theorem to derive sample complexity bounds for various UI operations. These bounds guide our algorithms when setting up the sample size.

C.1 Quantile Estimation

Assume that we have V vertical pixels in our display, and N rows in the table. Assume there is a sorted order on these rows, and define the (relative) rank of the ith row to be i/N. Let $D = \{i/N\}_{i=1}^{N}$ denote the set of rows (identified by rank). Assume that the user moves the scroll bar to pixel j. If we were computing quantiles exactly, scrolling to pixel j would render the screen whose top row has rank j/V. We relax this notion and view pixel $j \in [V]$ as representing a set of rows, whose rank lies in the interval $I(j) = (j/V - \varepsilon, j/V + \varepsilon)$. A valid output for our quantile estimation routine is to return any element from this range.

Our algorithm is to take a sample S of uniformly random rows from $[V]$ an return the element whose relative rank is closest to j/V.

THEOREM 2. *If $|S| \geq O(\varepsilon^{-2}\log(1/\delta))$, the above algorithm returns an element from $I(j)$ with probability $1-\delta$.*

Proof: Let $\mathscr{P}$ denote the uniform distribution on D. Let $\mathscr{F}$ denote the class of intervals $(a,b] \subseteq [0,1]$, this class has VC-dimension 2. Note that $\mu_{\mathscr{P}}(a,b] = b - a \pm O(1/N)$. We ignore the $1/N$ factor, since our assumption is that $N \gg V$.

To ensure that our algorithm returns a row from $I(j)$, it suffices that both the intervals $L = [0, j/V - \varepsilon]$ and $R = [j/V + \varepsilon, 1]$ adjoining $I(j)$ satisfy (1) for $\varepsilon/2$. If this is the case, then

$$\frac{|S \cap L|}{|S|} \leq j/V - \varepsilon/2, \frac{|S \cap R|}{|S|} \leq 1 - j/V - \varepsilon/2$$

This means that all the elements whose relative rank in S lie in the range $(j/V - \varepsilon/2, j/V + \varepsilon/2)$ must come from $(L \cup R)^c = I(j)$ The sample complexity follows by Theorem 1. □

In practice, we choose $\varepsilon = 1/(2V)$ so that the intervals $I(j)$ give a disjoint partition of $[0,1]$, which requires sample complexity $O(V^2)$ for constant probability of success.

C.2 Histograms and HeatMaps

Assume we have a table containing a set of rows D from a range $\mathscr{D}$ which is ordered. By mapping each element of D to its relative rank, we can identify D with a subset of points in $[0,1]$. The uniform distribution on D gives a distribution $\mathscr{P}$ on $[0,1]$. To draw a histogram, we divide $[0,1]$ into B equal intervals/buckets. Assume we have V vertical pixels in our histogram. For each $b \in [B]$, let $p(b)$ denote the fraction of the population in the bth bucket and let $p_{\max}$ denote the largest value.

A natural choice for the vertical range is $[0, p_{\max}]$, so that a bar reaching pixel j represents probability mass of $j \cdot p_{\max}/V$. If we knew each probability $p(b)$ exactly, we would snap that bar to the closest pixel, so that pixel j represents the interval $I_j = [(j-0.5)p_{\max}/V, (j+0.5)p_{\max}/V)$. Let us refer to this as the *ideal* histogram. Even in the ideal histogram, there is a *rounding error* of $p_{\max}/V$ in the rendering of $p(b)$: if the bar for bucket b has height j, then we know that $p(b) \in I(j)$ which has width $p_{\max}/V$.

We define an approximate histogram as one where pixel j represents probabilities from a slightly larger range. Concretely in a μ-approximate histogram, every bar with height j represents a probability in the range $\widehat{I}_j = [(j-0.5-\mu)p_{\max}/V, (j+0.5+\mu)p_{\max}/V]$. As long as $\mu < 0.5$, this ensures that every bar is within a pixel of its height in the ideal histogram.

Our algorithm for computing approximate histograms is the obvious one: we take a sample of n uniformly random rows from $[N]$ and use this to determine empirical probabilities $\hat{p}(b)$ for each bucket $b \in B$. We determine $\hat{p}_{\max}$ and then assign a bar of height j to bucket b where $j\hat{p}_{\max}/V$ is closest to $\hat{p}(b)$.

THEOREM 3. *With $n = O(V^2/(\mu p_{\max})^2 \log(1/\delta))$ samples, the above algorithm computes a μ-approximate histogram with probability $1-\delta$.*

Proof: As in Theorem 2, we identify D with a discrete subset of $[0,1]$, and each bucket with an interval. We take $\varepsilon = \mu p_{\max}/2V$ and compute the sample complexity by Theorem 1. With probability $1-\delta$, we have $|\hat{p}_{\max} - p_{\max}| \leq \varepsilon$. If bucket b maps to pixel j, it implies that

$$\hat{p}(b) \in [(j-0.5)\hat{p}_{\max}/V, (j+0.5)\hat{p}_{\max}/V]$$

hence $p(b)$ lies in the interval

$$[(j-0.5-\mu/2)\hat{p}_{\max}/V, (j+0.5+\mu/2)\hat{p}_{\max}/V] \subseteq \widehat{I}_j$$

so the claim follows. □

If we set μ and δ to some constants (say 0.1 and 0.01), the sample complexity by Theorem 1 is $O(V^2/p_{\max}^2)$. In the worst case, $p_{\max}$ might be as small as $1/B$ (where B is the number of buckets), although typically it is much larger. In practice, we have found that using CV^2 samples for constant C works well.

A closely related problem is that of computing the CDF of the distribution of values in a table. We think of each pixel representing a single bucket, where buckets are obtained by dividing the range into equal parts. But rather than the individual mass $p(b)$, we want to plot the cumulative sum $\sum_{i \leq b} p(b)$. To map this to the VC dimension setting, we identify bucket/pixel b with the interval $[0, b/V]$. Setting the accuracy to $1/2V$ is sufficient, since the vertical range is $[0,1]$. So by a similar argument to Theorem 3 above, $n = O(V^2/\log(1/\delta))$ samples suffice.

Similar bounds can be derived for 2-d histograms. In this case, there are sort-orders on two columns, which lets us map rows to a discrete subset of $[0,1] \times [0,1]$. Each bucket is an axis-aligned rectangle in $[0,1] \times [0,1]$, a class which has VC dimension 3. The desired additive accuracy is again roughly $1/V$ for 2-d histograms.

For heatmaps the way density is represented as a color is very important. It matters whether the colors are continuously varying from the background color, or there is an abrupt jump from 0 to 1. Color scales can also be linear or logarithmic. Linear color scales are fairly similar to histograms. Assume that the maximum probability for any bucket is $p_{\max}$. Assume a continuous color scale where there are $C \approx 20$ discernibly distinct colors, so each corresponds to an interval of length $p_{\max}/C$. A natural goal in sampling is that our estimation error should be roughly $p_{\max}/4C$ so that each bucket gets almost the same color as in the ideal heatmap. This can be achieved using $O(C^2/p_{\max}^2)$ samples. $p_{\max}$ can be as small as $1/HV$ where H and V are the number of horizontal and vertical pixels, though typically it is much higher.

Heatmaps where the color is on a log-scale are harder for sampling. Here we need a constant factor multiplicative approximation to each $p(b)$ rather than an additive approximation. This is harder to achieve via sampling, since it requires accurate estimates even for

very small probabilities, and is better done by a 1-pass streaming algorithm.

C.3 Heavy Hitters

In the heavy hitters problem, we are given a table with N rows, each holding a value from a domain D and a threshold $\alpha \in [0,1]$. Let $f(d)$ denote the number of occurrences of item $d \in D$ in the table. Our goal is to find all $d \in D$ that occur with frequency at least αN. The naive algorithm is to take a sample of size n and return all elements that occur with frequency at least $3\alpha n/4$.

THEOREM 4. *Let $n > \log(1/\alpha\delta)/\alpha^2$. With probability $1-\delta$, the algorithm above returns all i such that $f(i) \geq \alpha N$ and does not return any elements where $f(i) \leq \alpha N/4$.*

Proof: We say that an element is heavy if it occurs more than αN times, and light if it occurs fewer than $\alpha N/2$ times. The Chernoff-Hoeffding bound [83, Lemma 4.5] implies that the probability that a heavy element occurs in the sample less than $3\alpha n/4$ times is bounded by $\exp(-\alpha^2 n/16)$. A similar bound holds for light elements. To finish the argument, we wish to use the union bound. There are at most $1/\alpha$ heavy elements. There can be many ($\Omega(N)$) light elements, but we can group them together into groups so that each group (except one) occurs with frequency in the range $[\alpha N/4, \alpha N/2]$, and claim that no group occurs too frequently in our sample. This of course implies that elements in that group are also not too frequent. Now we apply the union bound, across at most $5/\alpha$ groups of light elements and $1/\alpha$ heavy elements. Overall, we want

$$\frac{6}{\alpha}\exp(-\alpha^2 n/16) \leq \delta$$

which is satisfied for our choice of n. □