Data Canopy: Accelerating Exploratory Statistical Analysis

Abdul Wasay Xinding Wei Niv Dayan Stratos Idreos

Harvard University

{awasay,weixinding,dayan,stratos}@seas.harvard.edu

ABSTRACT

During exploratory statistical analysis, data scientists repeatedly compute statistics on data sets to infer knowledge. Moreover, statistics form the building blocks of core machine learning classification and filtering algorithms. Modern data systems, software libraries, and domain-specific tools provide support to compute statistics but lack a cohesive framework for storing, organizing, and reusing them. This creates a significant problem for exploratory statistical analysis as data grows: Despite existing overlap in exploratory workloads (which are repetitive in nature), statistics are always computed from scratch. This leads to repeated data movement and recomputation, hindering interactive data exploration.

We address this challenge in Data Canopy, where descriptive and dependence statistics are synthesized from a library of basic aggregates. These basic aggregates are stored within an in-memory data structure, and are reused for overlapping data parts and for various statistical measures. What this means for exploratory statistical analysis is that repeated requests to compute different statistics do not trigger a full pass over the data. We discuss in detail the basic design elements in Data Canopy, which address multiple challenges: (1) How to decompose statistics into basic aggregates for maximal reuse? (2) How to represent, store, maintain, and access these basic aggregates? (3) Under different scenarios, which basic aggregates to maintain? (4) How to tune Data Canopy in a hardware conscious way for maximum performance and how to maintain good performance as data grows and memory pressure increases?

We demonstrate experimentally that Data Canopy results in an average speed-up of at least 10× after just 100 exploratory queries when compared with state-of-the-art systems used for exploratory statistical analysis.

1. INTRODUCTION

Data Science and Statistics. Many data science pipelines across different fields begin with a data exploration phase [75]. During this phase, data scientists develop an initial understanding of the data by using statistics to summarize variables within the data set, understand trends in variables, and correlate these trends with those of other variables [41, 58]. For instance, variance in seismic activity of an area represents how prone it is to earthquakes and correlations between seismic measurements across various sensors help to predict future patterns of seismic activity [77]. Moreover, statistics – such as mean, variance, and correlations – serve as building blocks of core machine learning classification and filtering algorithms such as simple linear regression, bayesian classification, and collaborative filtering [14]. Overall, statistical analysis forms the staple of data exploration across all fields [25, 32].

Repetitive Calculation of Statistics. Exploratory statistical analysis, a typically unstructured procedure, results in repetitive calculation of statistics. Every result provides data scientists with knowledge and cues for what to ask next or which model to try out. By query here we mean a request to compute a given statistic over a given data part. Different statistics are successively computed on the same part of the data or even the same statistics are recomputed with varying resolution and on data ranges (data portions) that overlap with previously accessed data ranges. Effectively an exploration session consists of numerous such repeated queries until a pattern is found [47]. Figure 1 shows different forms of such repetitive access patterns.

Repetition appears in various forms in various workloads. Figure 2(a) shows the repetition in two publicly available workloads: SDSS SkyServer [33] and SQLShare [45]. These workloads are composed of both handwritten and computer generated SQL queries. Up to 97 percent of the queries repeat at least once in SDSS. Queries repeat less frequently in the SQLShare workload, however, up to 55 percent of queries still target a non-distinct set of columns. Furthermore, studies show that repetition is higher in interactive exploratory analysis [48].

Data Science Tools and Statistics. Data scientists have a spectrum of tools available to them for exploratory statistical analysis. This spectrum, at one end, includes software libraries, such as NumPy [1] and Modeltools [34], with flexible functionality but no in-built data management. On the other end of this spectrum are highly optimized relational database systems, but with limited statistical functionality. Database connectors like SciDB-Py [69], MonetDB.R [61], and Psycopg [2] connect a database backend with a flexible language thereby providing a good compromise between flexibility and data management. Such connectors provide the major benefit of computing statistics inside the database system without having to move the data.

To get a sense of how modern systems behave during exploratory statistical analysis, we perform the following experiment. We use two data columns each with 100 million rows of doubles (unique, uniformly distributed). We simulate an exploratory analytics sequence by firing successively a series of queries to compute var-

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064051

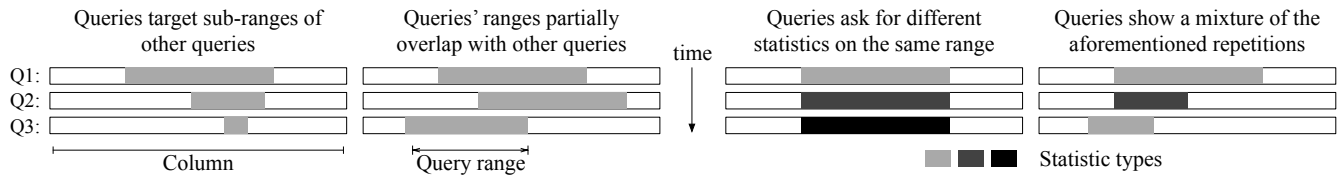
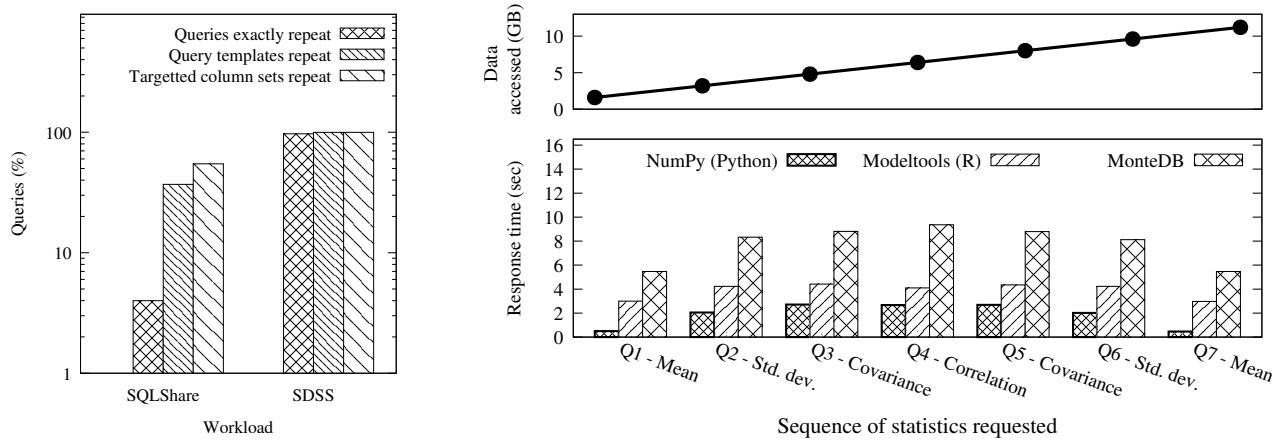


Figure 1: In exploratory statistical analysis, queries request for a given statistic on a given data range and show various forms of repetition.



(a) Exploratory workloads in the sciences exhibit high repetition in queries.

(b) Existing systems used in exploratory statistical analysis do not reuse computation and data access across statistical queries (both top and bottom parts use the same *x*-axis).

Figure 2: Data Canopy motivation: Existing systems always compute different statistical measures from scratch causing significant slowdown in the presence of repetitive exploratory workloads.

ious statistics. Results are shown in Figure 2(b) (the bottom part shows query response time and the top part shows the amount of data accessed; for MonetDB the statistics are computed inside the DBMS). The main observation is that as we fire more queries (as the *x*-axis evolves from left to right), all systems maintain a rather stable query response time; it fluctuates a bit depending on how computationally heavy each statistic is. Critically, this continues to hold even when, in the second half of the query sequence, we ask for exactly the same set of statistics again. This behavior is explained by the amount of data that each of these systems has to access (top part of Figure 2(b)). It is the same for all systems as they touch the same data but the important point is that accesses accumulate as we ask for more statistics. In turn, what this means is that every time data scientists want to explore a new statistic, to understand a different property of the data set, they have to incur the overhead of going over the whole data again.

Lost Opportunities. Repetitive workloads, on one hand, and the absence of a cohesive framework to store and reuse statistics on the other hand, result in sub-optimal performance: (1) No matter the degree of overlap in workloads, existing systems and data science tools always compute statistics from scratch; (2) No central framework exists to opportunistically or preemptively collect statistical measures to speed up the process of exploratory statistical analysis; (3) User queries and machine learning algorithms, which, directly or indirectly, compute and use statistics cannot share computation and data access.

As data sets continue to grow, calculating statistics from scratch each time during interactive exploratory statistical analysis becomes intractable. For instance, in the Earthscope project, an array of four hundred sensors continuously records seismic activity around the US, which alone results in about eighty thousand unique correlations [77]. In addition, the fully-sequenced human genomic data, composed of over three billion base pairs per individual, is projected to outgrow our ability to analyze it by 2025. According to an estimate analyzing two billion genomes per year in parallel will require a processing speed of two genomes per CPU hour [72], which already exceeds our current ability by three to four orders of magnitude [53]. Doing it repeatedly is completely unrealistic.

Data Canopy. In this paper, we take a step to address this problem by introducing Data Canopy. In Data Canopy statistics, alongside data, become first class objects within the data system. Data Canopy maintains a library of *basic aggregates* that can be used to synthesize statistics without repeatedly accessing base data. These basic aggregates, depending on the statistical measure being computed, can take multiple forms and can be reused in different ways. For instance, when computing standard deviation, Data Canopy stores the resulting basic aggregates: sum and sum of squares. The sum can later be reused to completely synthesize the mean and the sum of squares can later be reused as one of the ingredients for correlation coefficients across variables.

Individual basic aggregates within the library are computed and maintained at a granularity of a *chunk*. A chunk is the smallest portion of data (e.g., a collection of k contiguous values in a column) that Data Canopy maintains basic aggregates on. This allows reuse between queries that request statistics on overlapping or partially overlapping data. For instance, in a time series data set weekly correlations are synthesized from daily correlations. Also, in settings with limited amount of main memory, the chunk size can be used to adjust the tradeoff between memory requirement and the resolution of stored information.

Effectively, Data Canopy is a smart cache of the basic primitives of statistical measures. Data Canopy can be populated in different ways depending on the scenario: (1) In *offline* mode, Data Canopy is constructed over a specified part of the data set completely in advance; (2) In *online* mode, Data Canopy populates the library of basic aggregates incrementally online during query processing;

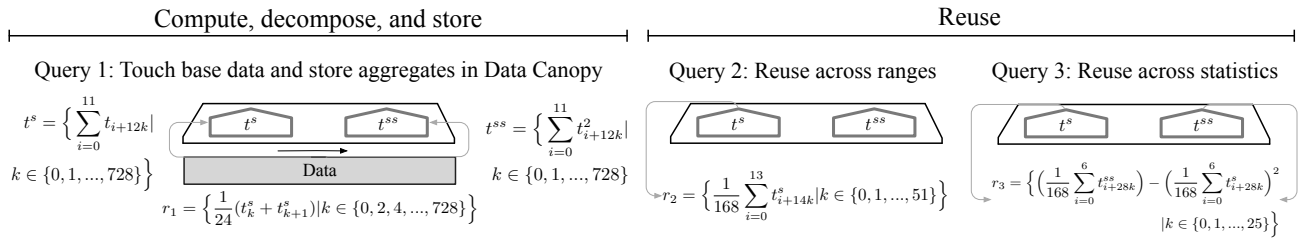


Figure 3: An example of queries that can reuse computation and data access through Data Canopy.

(3) In *speculative* mode, Data Canopy speeds up the population of the library of basic aggregates by speculatively creating and maintaining additional basic aggregates in addition to those required by active queries.

Contributions. Our contributions are as follows:

- We demonstrate that existing systems used for exploratory statistical analysis cause redundant data movement, which becomes a bottleneck as data grows.
- We propose Data Canopy, a smart cache tailored for exploratory statistical analysis. It computes and caches the basic primitives of statistical measures and then it can synthesize results for future queries without having to repeatedly go back to base data.
- We discuss the design space of Data Canopy in detail. We show how to break up statistics into basic aggregates and maintain them at an optimal granularity that enables efficient synthesis of other statistics during query time.
- We show how to store and maintain basic aggregates in a way that provides logarithmic query time for queries over arbitrary data portions and allows Data Canopy to be built and updated incrementally.
- We develop policies for various core scenarios so that data scientists may use Data Canopy both offline, i.e, when there is time to let Data Canopy scan the data to precompute the library of basic aggregates, and online, i.e., when there is no time to devote to preparation. Data Canopy can also opportunistically compute basic aggregates during query processing to speed up future queries.
- We show how to achieve a hardware conscious tuning of the chunk size to optimize read performance and how to react to memory pressure as data grows.
- We demonstrate that Data Canopy results in a speed up of $10\times$ in repetitive workloads compared to state-of-the-art systems currently used in exploratory statistical analysis.

2. DATA CANOPY

We now present Data Canopy in detail. Data Canopy allows data scientists to perform exploratory statistical analysis without having to repeatedly scan the entire base data.

The main idea is that Data Canopy breaks statistics down to basic aggregates. It caches and manages a library of basic aggregates so that incoming queries may use it to synthesize different kinds of statistics. Data Canopy can compute the library of basic aggregates in a single offline pass over the data. For dynamic scenarios with little idle time, Data Canopy incrementally computes the library of basic aggregates during query processing.

2.1 Example

First, we motivate and provide the core intuition of Data Canopy with an example before discussing the design. Consider the hourly temperature measurements collected by the National Centers for Environmental Information (NCEI) [3]. On this data set we build an instance of Data Canopy that is configured to work with three univariate statistics: mean, variance, and standard deviation. Figure 3 shows how Data Canopy processes a series of queries over this data set without having to always check the base data.

Query 1: The data scientist requests mean temperatures for each day. Data Canopy is initially empty i.e., there are no basic aggregates to utilize. For this query Data Canopy has to access base data and compute the daily mean temperatures (using 24 observations for each calculation). Data Canopy takes this opportunity to compute and store two types of basic aggregates: (1) basic aggregates that are immediately needed to synthesize statistics for the current query, and (2) basic aggregates that are not immediately needed, but can be computed from accessed data and then reused by other statistics. These basic aggregates are always maintained at a fixed granularity of a chunk. For ease of presentation, the chunk size is set to 12 in this example, i.e., one chunk corresponds to twelve hours (in practice Data Canopy autotunes the chunk size as we will discuss later on). The basic aggregates resulting from this query are shown under Query 1 in Figure 3. For every chunk of size 12, Data Canopy stores the set of sums (t^s), to be used for the current query, and the set of sums of squares (t^{ss}), that may be used by future queries (for example for standard deviation and variance).

Query 2: The data scientist requests mean temperatures for each week. This time the data scientist asks for the same statistic as requested in Query 1 but at a different granularity (weekly instead of daily). As shown under Query 2 in Figure 3, there is no need to access the base data again. Data Canopy already contains t^s, the sums of hourly temperatures for every 12 hours. It sums up 14 consecutive values of t^s to synthesize the result for each week.

Query 3: The data scientist requests variances in temperature for every two weeks. This time the data scientist asks for both a different statistical measure and at a different granularity (biweekly instead of weekly or daily). As shown under Query 3 in Figure 3, Data Canopy synthesizes statistics from basic aggregates, again, without accessing the base data. The variance of a set of observations x is given by Equation 1. Data Canopy thus uses t^s and t^{ss} to synthesize the result set r_3 for this query.

$$v_x = \Big(\frac{1}{N}\sum_{i=1}^{N} x_i^2\Big) - \Big(\frac{1}{N}\sum_{i=1}^{N} x_i\Big)^2 \quad (1)$$

Other Queries. Similar to the above scenarios, once Data Canopy stores the set of sums t^s for every 12 hours, and the set of sums of squares t^{ss} for every 12 hours, it can reuse these basic aggregates in four different types of query scenarios:

Statistics		Basic Aggregates				
Type	***Formula***	$\sum x$	$\sum x^2$	$\sum xy$	$\sum y^2$	$\sum y$
Mean (avg)	$\frac{\sum x_i}{n}$	■				
Root Mean Square (rms)	$\sqrt{\frac{1}{n} \cdot \sum x^2}$		■			
Variance (var)	$\frac{\sum x_i^2 - n \cdot \text{avg}(x)^2}{n}$	■	■			
Standard Deviation (std)	$\sqrt{\frac{\sum x_i^2 - n \cdot \text{avg}(x)^2}{n}}$	■	■			
Sample Kurtosis (kur)	$\frac{1}{n}\sum(\frac{x_i - \text{avg}(x)}{\text{std}(x)})^4 - 3$	■	■			
Sample Covariance (cov)	$\frac{\sum x_i \cdot y_i}{n} - \frac{\sum x_i \cdot \sum y_i}{n^2}$	■		■		■
Simple Linear Regression (slr)	$\frac{\text{cov}(x,y)}{\text{var}(x)}, \text{avg}(x), \text{avg}(y)$	■	■	■		■
Sample Correlation (corr)	$\frac{n \cdot \sum x_i \cdot y_i - \sum x_i \cdot \sum y_i}{\sqrt{n \cdot \sum x_i^2 - (\sum x_i)^2}\sqrt{n \cdot \sum y_i^2 - (\sum y_i)^2}}$	■	■	■	■	■

Table 1: Data Canopy synthesizes statistics from a library of basic aggregates.

Term	Description
c	Number of columns
r	Number of rows
h	Number of chunks
s	Chunk size (bytes)
v_d	Record size (bytes)
v_{st}	ST node size (bytes)
#	Cache line size (bytes)

Table 2: Data Canopy terms.

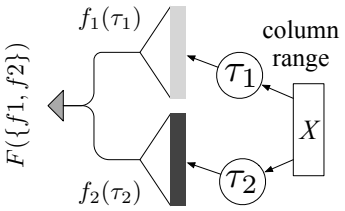


Figure 4: Decomposing Statistics.

i. *Across different data ranges*: daily mean of the first three days, daily mean of the last four days, etc.

ii. Across different data granularities: weekly mean, biweekly mean, etc.

iii. *Across different statistical measures*: daily standard deviation, daily variance, etc.

iv. *Across any combinations of i, ii, and iii*: weekly standard deviation, monthly variance, etc.

In the rest of this section, we discuss Data Canopy design concepts, data structures, and different policies that seamlessly enable the aforementioned degree of reuse.

2.2 Design Concepts

We now describe the core design concepts in Data Canopy.

Data and Query Range. We will use the concepts of data and query range throughout our discussion. We define a data range as a set of consecutive data items from a column or a set of columns. A query range is the data range over which a query requests statistical measures.

Basic Aggregates. Data Canopy breaks statistical measures into basic primitives. We call those primitives basic aggregates. We define a basic aggregate over a data range as a value that is obtained by first performing a transformation τ on every data item in that data range and then combining the results using an aggregation function f. Formally, for a given data range X, (with elements x_i) a basic aggregate can be represented as $f(\{\tau(x_i)\})$. In our running example, sum of squares t^{ss} can be represented as $f(\{\tau(x_i)\}) = \sum_i x_i^2$, where $\tau(x_i) = x_i^2$ and f is the sum function.

The transformation τ can be any operation on an individual data item. However, the aggregation function f has to be commutative and associative i.e., we should be able to break down and combine basic aggregates between sub-ranges (partitions of the data range). Formally, for any partition $\{X_1, X_2, \ldots X_n\}$ of a data range X, the following should hold:

$$f(X) = f(\{f(X_1), f(X_2) \ldots f(X_n)\}) \quad (2)$$

For instance, this property is satisfied by min, max, count, sum, and product functions on any given data range, whereas the median function does not satisfy this property.

Decomposing Statistics. Data Canopy defines a statistic S over a data range X as a function F of different basic aggregates:

$$S(X) = F(\{f(\tau(\{x_i\})\})$$

Figure 4 shows how statistic S (with function F) is mapped to two basic aggregates. The rationale behind representing statistics as a function of basic aggregates is twofold: First, various statistical measures share – and can reuse – basic aggregates. For instance mean, variance, and standard deviation all require the basic aggregate of sum over the target data. Second, a given basic aggregate over a certain data range (as a result of the property in Equation 2) can be further decomposed into sub-ranges. These sub-ranges can be combined together to synthesize that basic aggregate over any data range that contains those sub-ranges.

Table 1 shows how Data Canopy breaks down a set of widely used descriptive and dependence statistics into five basic aggregates. Effectively, Data Canopy is a smart cache. An alternative approach could be that we cache the result values of each individual statistic. However, we then lose the ability to reuse computation and data access between different statistics, despite clear overlaps. For instance, if instead of caching each of the basic aggregates corresponding to correlation, we cached just the final value, we will not be able to use that value to synthesize any of the other statistical measures mentioned in Table 1. Instead, we would have to access the data set again to compute the individual statistics.

In addition to the examples in Table 1, geometric mean ($\tau(x) = x, f(X) = \prod_i x_i$), harmonic mean ($\tau(x) = \frac{1}{x}, f(X) = \sum_i x_i$) and other descriptive and dependence statistics can be synthesized from basic aggregates. Over 90 percent of statistics supported by NumPy and SciPy [1], and over 75 percent of statistics supported by Wolfram [7] (a popular mathematical computational language) can be expressed in the aforementioned form i.e., they can be decomposed and expressed in terms of τ, f, and F.

Chunks. Data Canopy maintains basic aggregates at the granularity of a chunk – a logical partition of data that comprises of consecutive values from a data column. For every chunk, Data Canopy maintains a single value per basic aggregate type. In our example of hourly temperature data, a chunk size of 12 implies that for every statistical measure that Data Canopy computes, it caches each of the resulting basic aggregates over every 12 data values. This concept of chunk is essential to how Data Canopy enables reuse –

Options	Memory	Query/Update
ST per Data Canopy	$2 \cdot b \cdot c \cdot h - 1$	$O(\log b \cdot c \cdot h)$
ST per column	$2 \cdot b \cdot c \cdot h - c$	$O(\log b \cdot h)$
ST per statistic	$2 \cdot b \cdot c \cdot h - s$	$O(\log c \cdot h)$
ST per column per stat.	$2 \cdot b \cdot c \cdot h - b \cdot c$	$O(\log h)$

Table 3: Memory, access, and update cost of different configurations of segment trees (ST) storing b basic aggregates. The configuration used by Data Canopy (bottom) has the lowest query cost and memory usage.

reducing repeated data access – between different queries during exploratory statistical analysis.

As a result of chunking, queries of any data range larger than the chunk size can be synthesized directly from basic aggregates. Even in cases when the query range does not exactly align with the chunks, Data Canopy only needs to scan at most the two chunks at the edges of the requested query range. In a similar fashion, queries having partial range overlaps with previously computed chunks can also reuse basic aggregates. Mapping this concept to our running example, weekly and yearly variances in temperature can be synthesized from daily aggregates. Also, a query that requests the mean temperature over the last three weeks of a month, can reuse overlapping basic aggregates corresponding to the first two weeks.

Overall. Data Canopy is able to reuse previously computed basic aggregates to synthesize a wide set of statistics. As a concrete example, by storing just two basic aggregates of sum and sum of squares over five chunks in ten columns (a total of 100 values), Data Canopy can reuse this information across queries that target 2^5 possible combinations of chunks and request for up to four statistical measures – mean, variance, root mean square, and standard deviation – over any of these ten columns.

2.3 Data Structure

Data Canopy uses a set of segment trees to store basic aggregates. Segment trees support efficient aggregate queries over a data range without the need to access individual data items [26, 67]. This property is satisfied by storing, at every parent node, an aggregate of its two children. Segment trees in Data Canopy are implemented as binary trees. The Data Canopy catalog implemented as a hash table stores pointers to all segment trees.

Segment trees are well-suited as a data structure for Data Canopy. This is because to synthesize queries that request for statistics over a data range, Data Canopy only needs aggregates over chunks that fall within that data range, and not their actual values. Consider Query 1 in our running example. Data Canopy stores basic aggregates over 12 values (daily basic aggregates). A query that requests weekly standard deviation only needs sum and sum of squares over 14 consecutive basic aggregates, and not their actual values. This way, Data Canopy can synthesize statistics in time complexity which is logarithmic in the number of chunks involved.

Data Structure Configuration. For every basic aggregate kept for every column, Data Canopy maintains a separate segment tree. Every leaf of this segment tree stores a basic aggregate value corresponding to a chunk. An example layout of the Data Canopy data structure over a single column is shown in Figure 5. In this example, Data Canopy holds two basic aggregates (sum and sum of squares), using two separate segment trees, one for each basic aggregate.

By having a separate set of segment trees for every column, we ensure that the internal nodes of each segment tree contain no surplus nodes (i.e., those that maintain aggregates across columns or across statistical measures). As a result, the overall memory requirement of Data Canopy as well as the size of the individual segment trees is minimized. Also, since range queries are localized to a single column or a set of columns (for multivariate statistics) instead of the entire data set, we only have to search through a subset of the total segment trees, instead of one big segment tree corresponding to the entire data set. This arrangement still allows a data scientist or application to request for individual statistics and combine them in ways that make sense according to the domain and the data set. A comparison of the memory requirement and query cost of various possible configurations of segment trees is provided in Table 3. The configuration used in Data Canopy (bottom row of Table 3) has the lowest query cost and memory usage.

Flexibility. The separation of segment trees allows for maximum flexibility in dynamic and exploratory workloads. There is no need to construct or even allocate memory for the entire Data Canopy in advance. Instead, Data Canopy can easily be extended, by adding new segment trees, to cater for new columns or new basic aggregates.

Parallelism. The construction of Data Canopy can be aggressively parallelized as the process of calculating basic aggregates and storing them is an embarrassingly parallel one. To construct a univariate Data Canopy, the columns can be divided between the number of available hardware threads. Similarly, when constructing a multivariate Data Canopy, the segment trees for every combination of the columns can be built independently.

2.4 Operation Modes

Depending on hardware properties, data size, and latency requirements, Data Canopy can operate in one of three modes: *offline, online,* and *speculative*.

Offline. In the offline mode, Data Canopy is built in advance. This mode is useful when users know the data and statistical measures of interest a priori and they can also wait until Data Canopy is built before they pose their first query. The offline mode builds the library of basic aggregates fully for a set of rows, columns, and statistical measures specified by the user.

Online. In the online mode Data Canopy populates the library of basic aggregates incrementally online during query processing. For every incoming query, Data Canopy generates and caches the basic aggregates needed for this query if they do not already exist in the library. As more queries are being processed, the library of basic aggregates becomes more and more complete and can reduce data access costs for future queries with higher probability.

The online mode can be combined with the offline mode. For example, a user may generate any portion of the Data Canopy for any part of the data offline (or generate as much as idle time allows) and then during query processing, Data Canopy operates in online mode to fill in the rest of the missing pieces.

Speculative. In the speculative mode, Data Canopy takes full advantage of moving the data through the memory hierarchy to generate more knowledge than what is strictly needed for the active query. Every time it scans any part of the data set to answer a query, it builds segment trees for all univariate statistics. We show that this imposes a modest CPU and memory overhead for the current query, and Data Canopy potentially avoids having to rescan the data for future queries for other statistics – trading a modest CPU and memory overhead now for I/O benefits later on. For example, when Data Canopy answers a mean query in speculative mode, it also builds a segment tree for sum of squares so that it is possible to later efficiently synthesize the variance and standard deviation.

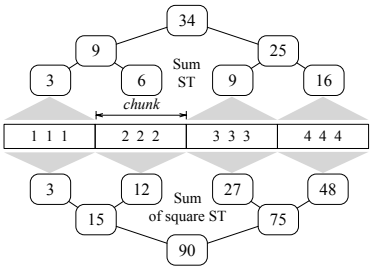
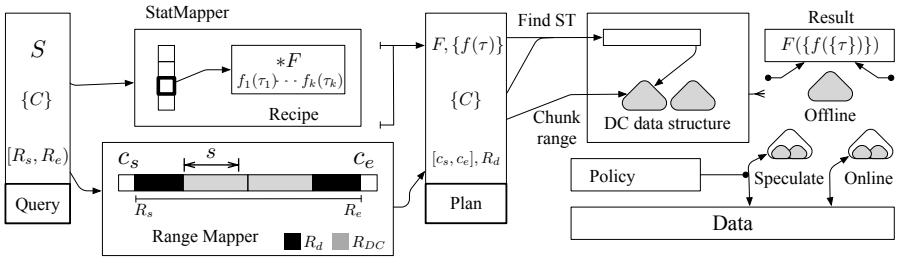


Figure 5: Example of the Data Canopy data structure with two segment trees (ST) and a chunk size of three.



Mapping the range of a query to a set of chunks and the requested statistic to a set of basic aggregates.

Based on the query and the policy, probing the DC data structure and materializing missing chunks

Figure 6: The lifecycle of a statistical query in Data Canopy.

2.5 Query Processing

We now explain how Data Canopy uses its library of basic aggregates to synthesize the results of statistical queries. We use terms from Table 2.

Query. In Data Canopy, a query is defined by the set $Q = \{\{C\}, [R_s, R_e), S\}$, where $\{C\}$ is the set of columns targeted by the query; R_s and R_e define the query range i.e., the two positions on the column set C on which a statistic is requested; and S is the statistical measure to be computed. From our running example, Query 2 (mean temperature for the third week) can be represented as $Q_t = \{C_t, [336, 504), \textit{mean}\}$. Figure 6 depicts the steps taken to process a query. The first step is to convert the query into a plan. To achieve this, the query range is mapped to a range of chunks, and the statistical measure is mapped to a set of basic aggregates.

Mapping Query Range to Chunks. Data Canopy first maps the query range $[R_s, R_e)$ to a set of chunks $[c_s, c_e]$, such that the whole query range is covered. This process is depicted on the left side of Figure 6, where the query range (shown in black and grey) is mapped to the corresponding chunks. Given the mapping, we can now distinguish between two parts of the query range. The first part of the query range R_{DC} (shown in grey) aligns perfectly with the boundaries of the existing chunks. In this case, Data Canopy can fully use the basic aggregates of these chunks to synthesize the result. The second part of the query range R_d (shown in black) at the two end-points of the query range might or might not align with the existing chunks. Data Canopy has to scan the two chunks at the end-points of the query range to compute basic aggregates for R_d. We call this part of the query range that always requires access to base data the residual range. When Data Canopy operates in online mode, it may be that it has to access more than two chunks so as to populate any missing chunks in any part of the query range, not just at the end points.

Mapping Statistic to Basic Aggregates. The next step is to map the requested statistical measure S to the corresponding set of basic aggregates $\{f(\tau)\}$ and a function F to combine these basic aggregates. This is achieved by the *StatMapper* as shown in Figure 6. For every statistical measure supported by Data Canopy, the StatMapper stores a complete *recipe* to synthesize that statistic from basic aggregates.

The StatMapper is implemented as a hash table, where the keys are identifiers of statistical measures and each key corresponds to a recipe. The recipe is a data structure that contains a list of basic aggregates $\{f(\{\tau\})\}$ required to synthesize the statistical measure S as well as a pointer to a function that operates on and combines the basic aggregates as defined by F. Overall, Data Canopy converts a query Q into a plan P, making the following set of mappings:

$$\{\{C\}, [R_s, R_e), S\} \rightarrow \{\{C\}, [c_s, c_e], R_d, \{f(\{\tau\})\}, F\}$$

Evaluating the Plan. The plan is passed on to the evaluation engine, where the result is synthesized based on the current policy and state of Data Canopy (right side of Figure 6).

If Data Canopy is operating in the offline mode, all basic aggregates have been precomputed and there is no need to touch the base data except to evaluate the residual range R_d. In this mode no new basic aggregates are added as a result of query processing. In the online and the speculative mode, some of the required basic aggregates (for some chunks) might not be computed and stored already. In such cases, Data Canopy accesses base data to evaluate basic aggregates on those chunks, and they are stored in Data Canopy. Finally, when all basic aggregates required for the current query are fetched and/or materialized, they are passed to function F to generate the result.

2.6 Analyzing Query Cost

We formalize the cost of answering a query when both Data Canopy and data fit in memory (we model the out-of-memory cost in §2.8). This cost is modeled in terms of the amount of data accessed (cache lines).

We consider a query q for a statistic S over a data range. The statistic S is defined over k different columns, and it is composed of b total basic aggregates i.e., it accesses b segment trees. For instance, in the case of a variance query, $b = 2$ (sum and sum of squares) and $k = 1$ (univariate statistic), whereas for a correlation query $b = 5$ (sum and sum of squares of both columns and sum of products) and $k = 2$ (bivariate statistic).

Let C_{syn} be the cost of answering query q. This cost is divided in two parts: (1) probing b segment trees, and (2) scanning the residual ranges of k columns. We denote these costs as C_{st} and C_r respectively. The total cost is:

$$C_{syn} = C_{st} + C_r$$

First, we model C_{st}. To answer a query q, Data Canopy traverses b segment trees. The number of leaves in each segment tree is $\frac{r \cdot v_d}{s}$, where r is the number of rows, v_d is the record size (in bytes), and s is the chunk size (in bytes). Moreover, the cost of probing a segment tree with n leaves is at most $2 \log n$ cache line reads [85] (as a node fits in a cache line). Hence, we can express C_{st} as follows:

$$C_{st} = 2 \cdot b \cdot \log_2 \left(\frac{r \cdot v_d}{s} \right) \tag{3}$$

We now model C_r. A query on k columns has to scan at most $2k$ chunks i.e., at the end points of the query range. The cost of scanning a chunk is $\frac{s}{\#}$. We get the following formula for C_r:

$$C_r = \frac{2 \cdot k \cdot s}{\#} \tag{4}$$

Using Equation 3 and 4, the total query cost becomes:

$$C_{syn} = \frac{2 \cdot k \cdot s}{\#} + 2 \cdot b \cdot \log_2 \left(\frac{r \cdot v_d}{s} \right) \tag{5}$$

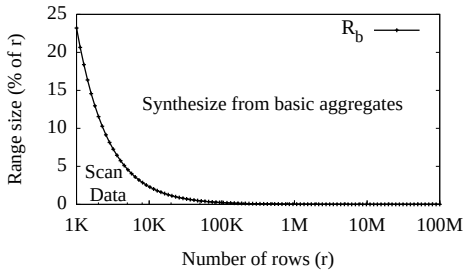


Figure 7: As the number of rows in the data set increases, a greater proportion of the total queries is answered through basic aggregates.

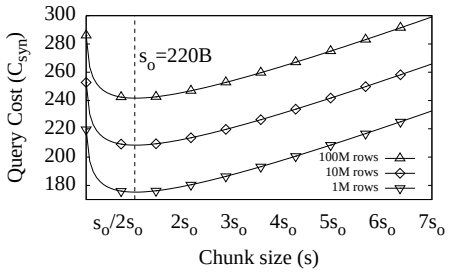


Figure 8: Query cost, a convex function of the chunk size, is minimized at the optimal chunk size s_o. Here #=64B, b=5, and k=2, s_o = 220B.

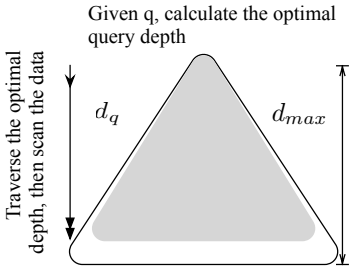


Figure 9: For each query, Data Canopy traverses the optimal depth d_q of the segment trees.

For simplicity of presentation, here we do not distinguish between the cost of a cache miss (traversing the linked segment trees) and a cache hit (scanning a sequential residual range). We study the effects of these hardware dependent parameters when we tune and verify the chunk size in Appendix E.

Synthesize or Scan. For queries with a small range, Data Canopy directly scans the data if this results in a smaller query cost compared to traversing the segment trees and synthesizing the answer. We describe below how this optimization decision is made.

The cost of scanning the full query range of size R, C_{scan} can be expressed as:

$$C_{scan} = \frac{R \cdot v_d}{\#} \tag{6}$$

Now we calculate the boundary query range size R_b, where C_{scan} becomes equal to C_{syn}. Below R_b, answering the query by scanning the complete query range is faster than synthesizing it from basic aggregates. Using Equation 5 and 6, we get:

$$R_b = \frac{2 \cdot k \cdot s}{v_d} + \frac{2}{v_d} \cdot \# \cdot b \cdot \log_2\left(\frac{r \cdot v_d}{s}\right) \tag{7}$$

Data Canopy answers a query with range size R from basic aggregates when $R > R_b$, otherwise it answers the query by scanning the full query range. Figure 7 shows how R_b (as a percentage of the number of rows r) decreases as r increases. This shows that as the number of rows in the data set increases a greater proportion of total queries is answered through basic aggregates. Here $\# = 64B$, $b = 5$, $k = 2$, and $v_d = 4B$.

2.7 Selecting the Chunk Size

We now explain how Data Canopy selects the chunk size so as to optimize query performance.

Optimal Chunk Size. The chunk size has opposite effects on the cost of scanning the residual range C_r and the cost of traversing segment trees C_{st}. Increasing the chunk size, results in an increase of C_r as the residual range increases. On the other hand, increasing the chunk size decreases C_{st} as the size of segment trees shrinks. As a result, C_{syn} is a convex function of the chunk size and has a global minimum i.e., there is an optimal chunk size s_o that optimizes overall query performance. The convex behavior of the query cost is shown in Figure 8 ($\# = 64B, b = 5, k = 2$). To obtain a closed-form expression for the optimal chunk size s_o, we differentiate C_{syn} with respect to s and equate the derivative to zero:

$$s_o = \frac{b \cdot \#}{k \cdot \ln 2} \tag{8}$$

The optimal chunk size s_o depends only on properties of the hardware (i.e., cache line size) and the type of requested statistic (i.e., the ratio between the number of segment trees and the columns that are scanned for the residual range). This is because the optimal chunk size strikes a balance between the number of cache lines accessed when scanning the base data (for the residual range) and when traversing the segment trees.

Optimal Chunk Size and R_b. Observe from Equation 7 that $s < R_b, \forall r \geq s$. In other words, any chunk size (including the optimal chunk size s_o) is always smaller than the boundary range size R_b below which a given query is answered by scanning the range. A corollary of this observation is that independent of the workload the chunk size should not be below s_o. This is because Data Canopy will answer any query with a smaller range size than s_o by directly scanning the range instead of traversing the segment trees (because this is faster i.e., it incurs fewer cache line reads).

Selecting the Chunk Size. By default, Data Canopy sets the chunk size s_{DC} to the lowest value of the ratio $\frac{b}{k}$. This value is 1 (for b=k=1) and allows Data Canopy to store just enough information (enough depth in the segment trees) to be optimal for queries that access the least amount of segment trees (e.g., mean, max, min etc.). Hence, to set the default chunk size, Data Canopy needs no prior knowledge of the workload or the data.

Workload Adaptivity. To ensure optimal performance for queries with $\frac{b}{k} > 1$ (i.e., those that access more than one segment trees), Data Canopy makes an adaptive decision and traverses shorter paths in the segment trees. This strategy is shown visually in Figure 9. Given a query q, Data Canopy analytically computes the optimal chunk size for this query s_q using Equation 8. Then it calculates the optimal depth of the segment tree for q:

$$d_q = \log_2\left(\frac{r \cdot v_d}{s_q}\right)$$

Data Canopy goes only as deep as d_q in the segment trees, and then scans the residual range (now up to a size of $2 \cdot k \cdot s_q$). This strategy ensures that each query achieves optimal performance by minimizing the data (cache lines) it has to read.

Overall, Data Canopy builds segment trees with a chunk size that guarantees optimality for queries that need to access a single segment tree only (i.e., d_{max}) and can afford to do more cache misses going all the way to the leaves of the segment tree. For queries that will access more segment trees, though, (and thus they will incur more cache misses) Data Canopy adaptively gets out of the segment tree traversal sooner (i.e., at d_q) reverting on sequentially scanning more data chunks and thus achieving an optimal balance tailored to each individual query. This optimization comes from the fact that segment trees are binary trees and every node we read when traversing the tree leads to a cache miss. As such there is a point when reading a cache line full of useful data (when scanning data chunks) becomes better than traversing a binary tree. Other directions, one may explore here, as alternatives to the optimization

we propose, is the study of a more cache conscious layout of the segment trees where every cache miss would bring a cache line full of useful tree data.

Memory Requirement. Data Canopy's memory requirement depends on: (i) the types of statistical measure it maintains, (ii) the chunk size, and (iii) the data size. For a given set of statistics S, we define the Data Canopy footprint $\mathscr{F}(S)$ as the number of segment trees per column required to synthesize S on the entire data set[1]. The size (in bytes) of a full segment tree with the optimal chunk size s_o and node size v_{st} is given by $v_{st} \cdot (2 \cdot \frac{r \cdot v_d}{s_o} - 1)$. Hence, the total size of a complete Data Canopy (in bytes) on c columns is:

$$|DC(S)| = c \cdot v_{st} \cdot (2 \cdot \frac{r \cdot v_d}{s} - 1) \cdot \mathscr{F}(S) \quad (9)$$

2.8 Out-of-Memory Processing

Now we introduce a three-phase eviction policy that maintains good performance guarantees as the data size and the size of Data Canopy exceeds main memory capacity. The high level idea is that Data Canopy maintains a cache of data pages, which are evicted when there is memory pressure and reloaded if needed. Similarly, parts of Data Canopy are also evicted and reloaded if needed. This policy captures both the case when data does not fit in memory and the case when Data Canopy does not fit in memory.

Phase 1. During the first phase, as main memory runs out, Data Canopy shrinks horizontally by removing one layer of leaf nodes from every segment tree in a round-robin fashion. This is equivalent to doubling the chunk size. Both data and Data Canopy still fit in main memory, and so the system maintains good performance (i.e., query processing is in the order of hundreds of microseconds). If there is more memory pressure and the chunk size exceeds the size of a page (4KB to 64KB), Data Canopy stops shrinking and moves on to Phase 2.

Phase 2. During Phase 2, Data Canopy maintains data pages in memory only as a cache of frequently accessed data. It evicts data pages from main memory using an LRU policy. Query cost remains low since each query has to touch at most $2k$ pages to scan the residual range, where k is the number of columns referenced by a query. For example, a correlation query needs to access at most two columns and thus touches at most four pages, which takes approximately 40 ms on modern disks. Moreover, for frequently accessed chunks, the cache prevents a query from going to disk.

Phase 3. In the extreme case, when none of the data can fit in memory, we reach the scenario, where parts of Data Canopy also need to be evicted. In this case, Data Canopy evicts whole segment trees using an LRU policy. These segment trees are spilled to disk and reloaded if needed. To make it easy when reloading segment trees from disk that may refer to potentially dirty chunks (updated), we keep an in-memory bit vector for each segment tree, which marks dirty chunks (1 bit per chunk). If memory pressure continues, bit vectors are also dropped along with the on-disk segment trees.

Offline Mode and Memory Pressure. When Data Canopy is set to offline mode it is given a set of data (row and columns) and a set of statistics to be precomputed. Data Canopy first computes the overall memory footprint that the resulting structure will have and if it exceeds available memory, Data Canopy has to operate immediately in Phase 3. Before doing so, Data Canopy first gives the user a warning and option if they want to reduce the amount of data or statistics to be included so that it fits in the memory budget. Otherwise, Data Canopy proceeds in Phase 3.

[1]For a complete discussion of the Data Canopy footprint and composability of statistics look at Appendix A and B.

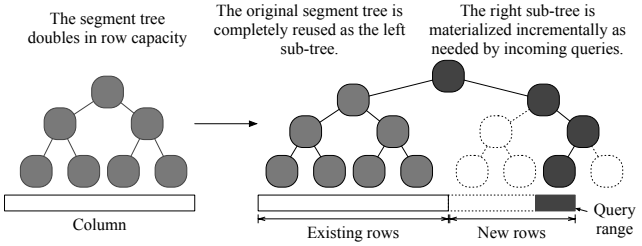


Figure 10: Data Canopy adaptively handles new data (rows).

2.9 Updates

We now discuss how Data Canopy handles insertions, updates, and deletes. Data Canopy handles updates incrementally to avoid overheads during online exploration.

Inserting Rows. When new rows are inserted and the new total number of rows exceeds the existing capacity of Data Canopy, then Data Canopy needs to expand. It does so by doubling the capacity of its segment trees without doubling the size immediately. This means that a root is added in each segment tree with the previous root as a left child and a new empty right child (and subtree). This results in effectively no immediate memory overhead. Data Canopy then populates the new right sub-tree adaptively only when and if the new rows are queried. This process is shown in Figure 10.

Inserting Columns. When a new columns is added, Data Canopy needs to simply add this column in its catalog. Given that columns are treated independently there is no further complexity resulting from the addition of a new column. As data in the new column is queried, Data Canopy allocates segment trees for this column and then populates them incrementally.

Updating Rows. When a record x at row r of column c is updated, Data Canopy first retrieves the old value x_{old} of x and uses it along with the new value x_{new} of x to update all segment trees that involve column c. For each segment tree, Data Canopy looks up the basic aggregate y_{old} for the chunk where row r resides, and it updates it as follows[2]: $y_{new} = y_{old} - \tau(x_{old}) + \tau(x_{new})$.

Assuming a univariate segment trees on column c, the cost of updating them is $a \cdot \log_2 \frac{r \cdot v_d}{s}$ (where $\log_2 \frac{r \cdot v_d}{s}$ is the depth of the segment trees). Moreover, assuming b bivariate segment trees on column c, the cost of updating them is $b \cdot \log_2 \frac{r \cdot v_d}{s} + b$. The additive b term derives from the fact we need to fetch one value from another column per segment tree to adjust the sum of products. The overall update cost C_{update} is:

$$C_{update} = 2 \cdot (a+b) \cdot \log_2 \frac{r \cdot v_d}{s} + b \quad (10)$$

Deleting Rows. Data Canopy deletes rows in-place using a standard technique for fixed-size slotted pages, where the granularity of a page is the chunk. Each chunk has a counter that keeps track of the number of valid rows in a chunk, and the valid rows are placed first in the chunk. When a row is deleted, we replace each deleted value x_{old} with the last valid value in the chunk, and we decrement the counter.

To update the segment trees, we probe all of them for the basic aggregate for the chunk of the deleted row and update it as follows[3]: $y_{new} = y_{old} - \tau(x_{old})$. In addition, we maintain one *invalidity segment tree* per table that keeps track of the number of invalid entries per chunk for subsequent statistical queries, as we can no longer

[2]More generally, we update y using the aggregation function F and its inverse F^{-1} as follows: $y_{new} = f\left(f^{-1}\left(\tau(x_{old}), y_{old}\right), \tau(x_{new})\right)$.

[3]More generally, we apply: $y_{new} = f^{-1}\left(\tau(x_{old}), y_{old}\right)$.

Workload	Column Dist.	Range Size	Repetition
U	Uniform	*Unif(5,10) %*	low
Z	Zipfian	*Unif(5,10) %*	moderate
U_+	Uniform	Zoom-in	high
Z_+	Zipfian	Zoom-in	very high

Table 4: Evaluation workloads.

assume that each chunk is full. The cost model is the same as for updates with one more additive term of $2 \cdot \log_2 \frac{r \cdot v_d}{s}$ for updating the invalidity segment tree: $C_{delete} = C_{update} + 2 \cdot \log_2 \frac{r \cdot v_d}{s}$.

3. EXPERIMENTAL ANALYSIS

We now demonstrate that Data Canopy accelerates statistical analysis and machine learning algorithms.

Experimental Setup. All experiments are conducted on a server with an Intel Xeon CPU E7-4820 processor, running at 2 GHz with 16 MB L3 cache and 1 TB of main memory. This server machine runs Debian "Jessie" with kernel 3.16.7 and is configured with a hard disk of 300GB operating at 15KRPM. We implemented Data Canopy from scratch in C++ compiled with gcc version 4.9.2 at optimization level 3. The current prototype supports three univariate statistics: mean, variance, and standard deviation; and two bivariate statistics: correlation, and covariance.

We compare the performance of Data Canopy with two widely used statistical packages: NumPy [1] in Python and Modeltools [34] in R. Also, we show how Data Canopy compares to MonetDB [16]. In addition to these systems, we compare Data Canopy against our own statistical system *StatSys*. StatSys shares the code base with Data Canopy, but it has none of the design concepts that allow Data Canopy to synthesize statistics from basic aggregates; instead, it needs to fully compute each query from scratch.

Benchmark. There are no standard benchmarks for exploratory statistical analysis. To test Data Canopy we develop a benchmark that captures a wide range of core scenarios and stress tests Data Canopy's capability to reuse data access and computation.

We generate exploratory statistical analysis pipelines as sequences of queries. Each query requests to compute a statistical measure on a range over a data column (or a set of data columns for multivariate statistics). The benchmark consists of four distinct workloads generated by varying two parameters: the probability with which queries are distributed over columns and the distribution of query range sizes. These workloads are summarized in Table 4. We investigate two different distributions of queries over columns: column-uniform (U and U_+) and column-zipfian (Z and Z_+). In the column-uniform workloads, queries are equally divided between all columns. In the column-zipfian workloads, queries are divided over columns conforming to the zipfian distribution (s=1) i.e., the column with the highest number of queries has twice as much queries in the workload as compared to the column with the second highest number of queries.

Similarly, we investigate two different distributions for the query range sizes. In the range-uniform workloads (U, Z), the range sizes are uniformly distributed between 5 and 10 % of the total column size. The range-zoom-in workloads (U_+, Z_+) emulate a case where data scientists progressively zoom into the data set increasing the resolution at which statistics are computed. In this case, the range size follows a sequence, where the first query is over an entire range. All subsequent pairs of queries divide the range of previous queries into two equal parts, then compute statistics on both. Then we randomly pick one of these parts to continue doing the same. For example, zoom-in over a range of size 100 can be the sequence: $\{[0,100), [0,50), [50,100), [50,75), [75,100)$ ... $\}$.

These workloads allow us to test Data Canopy with different kinds of repetition (similar to those presented in Figure 1). They map to patterns followed by data scientists during data exploration: The initial phase of exploratory analysis, often classified as the foraging phase [12, 64], exhibits patterns similar to column-uniform workloads. This is when data scientists compute statistics uniformly over multiple columns. Over time, the analysis focuses on a smaller set of columns (column-zipfian workloads), and requests for more detailed information (range-zoom-in workloads) [12, 64]. Moreover, the size of the data sets we use is derived from real world data sets (for a characterization of these data sets see Appendix C).

3.1 Reuse in Exploratory Statistical Analysis

In our first experiment we compare Data Canopy against state-of-the-art systems and we demonstrate its ability to reuse data access and computation. We set-up this experiment as follows: The data set contains 40 million rows and 100 columns. Each column is populated with double values randomly distributed in $[-10^9, 10^9)$. The total data size is 32GB. Data Canopy is automatically configured with the optimal in-memory chunk size. For our experimental system, this results in a chunk size of 256 bytes or 32 data values (in Appendix E we verify the chunk selection model). Data Canopy operates in the online mode, which provides an apples-to-apples comparison across all systems as it assumes no preprocessing steps.

Figure 11 shows the results for all four workloads. Each one of the four graphs in Figure 11 corresponds to one of the workloads in Table 4. Each graph depicts the evolution of the query performance (response time on the *y*-axis) as the query workload evolves, i.e., as we run more exploratory queries (*x*-axis). In total we run 2000 queries for each workload. Each graph shows the performance of NumPy, R, MonetDB, and Data Canopy.

The main observation across all graphs in Figure 11 is that while all state-of-the-art systems maintain a relatively constant behavior across all workloads, Data Canopy improves as it processes more queries. The *y*-axis is logarithmic and depicts the response time per query. For example in Figure 11(a) after just a hundred completely uniform queries, the average response time of Data Canopy is 1.9× lower than NumPy and 11.4× lower than MonetDB. After 2000 queries, the performance improvement per query goes up to 6.7× and 34.5× respectively. Thus, in most cases Data Canopy results in an overall benefit (during an exploration path, i.e., over a sequence of queries) of multiple orders of magnitude. The longer the exploration path the bigger the benefit.

In addition, Data Canopy is faster than all other systems even for the very first query across all workloads in Figure 11. This is because contrary to NumPy and R, Data Canopy is a tailored C++ implementation for statistics. MonetDB is a performant analytical system but it is not tailored for statistics.

Similar observations hold for Figures 11(c) and 11(d) where the workloads exhibit zoom-in patterns. In these workloads, the range size decreases by half after the first 500 queries. Then, it decreases by half every 1000 queries. This constant decrease in range sizes is reflected in the response times of all systems. In other words, all systems can improve nearly linearly to the size of the range on which statistics are computed. This is because they simply do computations on fewer data items. On the other hand, Data Canopy improves drastically by being able to reuse previous data accesses and computations. For all queries after the first 500 queries, the average response time goes down to sub-milliseconds. Even during the first 500 queries, there is a continuous sharp improvement in Data Canopy's response time. In both workloads, Data Canopy is

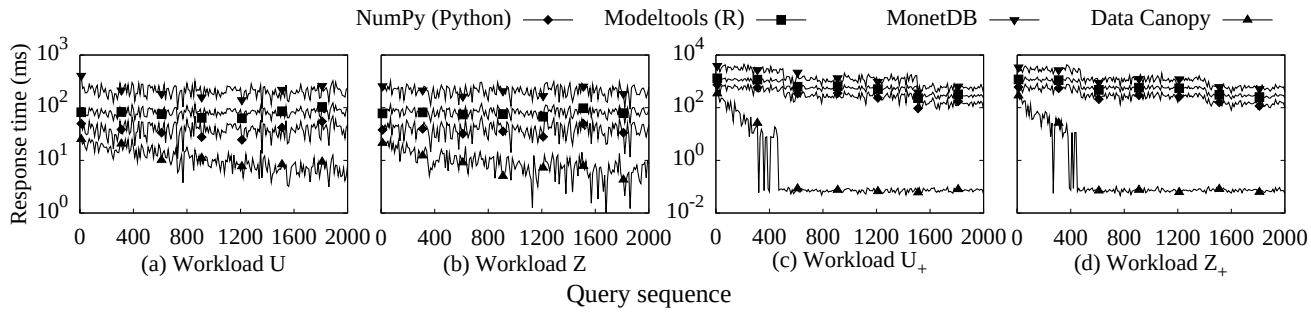


Figure 11: Data Canopy, in online mode, out performs state-of-the-art systems across a variety of workloads for exploratory statistical analysis by being able to incrementally improve its performance and minimize data access.

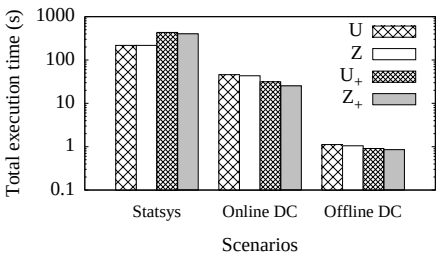


Figure 12: Online and offline Data Canopy result in one and two orders of magnitude improvement respectively.

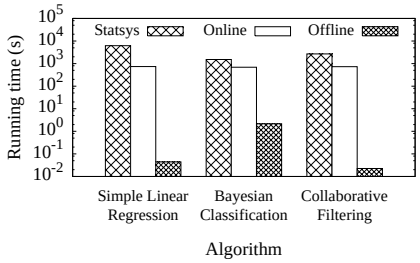


Figure 13: Data Canopy accelerates core machine learning classification and filtering algorithms.

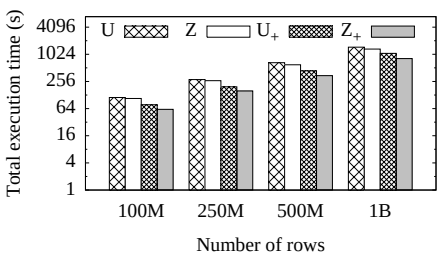


Figure 14: Data Canopy scales almost linearly with the number of rows in the data set for all workloads.

completely built at the end of the first 500 queries, and all future queries are directly synthesized from the basic aggregates within Data Canopy.

For all systems and for all these experiments we make sure that all data is hot in memory before we query it. This is the least favorable scenario for Data Canopy as its goal is to reduce data access costs.

Data Canopy Scenarios. Next we evaluate the offline and online modes of Data Canopy. In addition, we compare against StatSys, which effectively uses the Data Canopy code to compute statistics but does not cache and reuse basic aggregates.

The set-up of this experiment is exactly the same as before. The results are shown in Figure 12. This time we report the cumulative response time to run all queries. For all workloads Data Canopy results in significant benefits over the no reuse approach of StatSys (up to one order of magnitude i.e., 4.7× to 15.8×). If we can allow to precompute the library of basic aggregates up front this brings yet another benefit of two orders of magnitude (194× to 470.8×). In this scenario all queries are directly synthesized from Data Canopy (each query may at most scan two chunks at the boundaries of its range). Overall, the improvement is bigger for range-zoom-in workloads (U_+ and Z_+). This is because for these workloads the first query on every column results in a complete scan, due to which basic aggregates required for future queries on that column are already computed. Overall, Data Canopy is effective in both online and offline mode bringing drastic improvements in response time.

3.2 Accelerating Machine Learning

We now show how Data Canopy accelerates core machine learning classification and filtering algorithms. Specifically we study linear regression, bayesian classification, and collaborative filtering [14]. All three algorithm can utilize statistics (basic aggregates) cached in Data Canopy as primitives. The set-up is the same as in previous experiments (40 million rows and 100 columns) and we run each of the algorithms on the entire data set as follows: (i) Simple linear regression is ran on all pairs of columns, (ii) A gaussian naive bayes classifier is trained on the entire data set. In this case, the rows in the data set are divided between 40 different classes (one million samples per class), (iii) Collaborative filtering (using correlation as the similarity measure) is ran on the entire data set.

Figure 13 shows the performance of these three machine learning algorithms with Statsys (brute force), online, and offline Data Canopy. We observe that online Data Canopy (no preprocessing step) results in up to 8× improvement. This is because running these algorithms results in repetitive calculation of statistics. Furthermore, if there is enough idle time to build Data Canopy offline, we observe up to six orders of magnitude improvement in running time for simple linear regression and collaborative filtering and three orders of magnitude improvement for bayesian classification. The lower improvement for bayesian classification is due to the fact that we have to compute statistics for every class in the data set (i.e., 40 times more queries and each query results in scan of up to two chunks per column at the end-points of the query range).

3.3 Scalability

Here we show that Data Canopy scales with the number of columns and rows in the data set. Appendix D also offers a discussion on how Data Canopy scales with hardware contexts and queries.

Scaling with Number of Rows. First, we show how Data Canopy scales when we increase the number of rows in the data set. The set-up is the same as in previous experiments. This time we vary the rows from 100 million to one billion.

Figure 14 reports the results. It depicts the cumulative time to run all four workloads. As we increase the number of rows from 100 million to 250 million, the total execution time increases by 2.51x (average across all workloads) i.e., an approximately linear increase in execution time. As we double the number of rows beyond 250 million, the trend diverges slightly from a linear trend. The increase in cumulative response time as we increase the num-

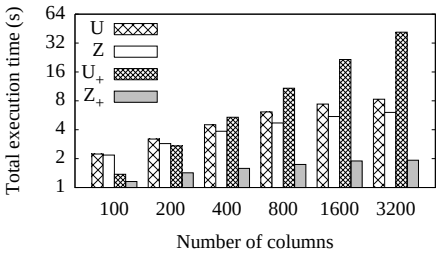


Figure 15: Data Canopy scales with the number of columns resulting in sub-linear increase in query execution time.

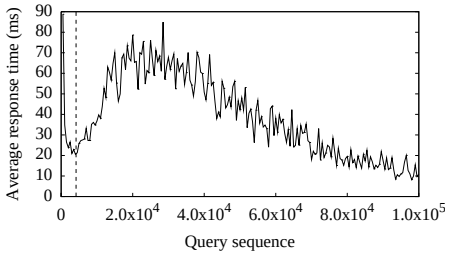


Figure 16: Data Canopy gracefully handles memory pressure, keeping query processing time within an interactive range.

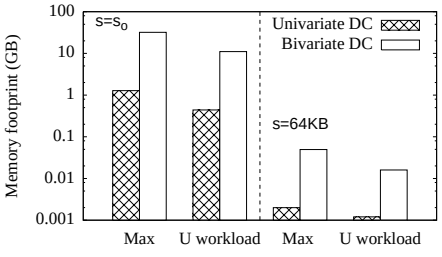


Figure 17: Under memory pressure, Data Canopy can vary its chunk size between the memory-optimized and disk-optimized size.

ber of rows from 250 million to 500 million and from 500M to 1 billion is 2.26x and 2.3x respectively. This super-linear increase in cumulative response time is due to the fact that with more rows, the size of the query range (unif(5,10)% of r) increases. This results in more chunks being added to the Data Canopy data structure, for every query that is executed. The overhead of adding these chunks results in this super-linear increase in the overall response time.

Scaling with Number of Columns. Now, we show how Data Canopy scales as we vary the number of columns from 100 to 3200. In this experiment, the number of rows is fixed to one million.

Figure 15 reports the cumulative time to run all four workloads. As we double the number of columns, we see an average increase of 1.68x and 1.22x in the total execution time for the uniform (U and U_+) and zipfian (Z and Z_+) workloads respectively. In all cases the execution time increases in a sub-linear fashion. For uniform workloads there is a higher increase in the total execution time because they target all columns equally and it takes longer to populate the library of basic aggregates. For the zipfian workloads, since the columns are targeted following a zipfian distribution, increasing the number of columns does not substantially affect the overall execution time – columns that are frequently accessed will have their corresponding library of basic aggregates completely materialized.

Overall, Data Canopy scales in a robust way, being able to absorb the increased amount of rows and columns.

3.4 Handling Memory Pressure

We now demonstrate that Data Canopy can gracefully handle memory pressure. For this experiment we allow a memory budget of 8GB. The size of the data is set to 7.2 GB (90 columns, 10 million rows, 8 bytes record size). This means that initially the entire data set fits in main memory. Data Canopy operates in online mode which means that initially it has zero memory footprint and it grows as more queries arrive. We run a sequence of queries from the U workload. This implies that Data Canopy incrementally materializes new segment trees, increasing memory pressure.

Figure 16 shows how the average response time of Data Canopy evolves as memory pressure increases. The dotted line depicts the point beyond which Data Canopy operates in Phase 2 of the out-of-memory policy i.e., some data is now accessed from disk. We observe that as Data Canopy enters Phase 2, there is an initial increase in query response time. This is because Data Canopy is still being built, and every query may result in a scan of data on disk. However, as the query sequence evolves and Data Canopy materializes further, the query response time decreases. Now, Data Canopy scans at most two chunks per query.

In Appendix F, we provide a study of how Data Canopy behaves when the memory pressure is due to data size.

3.5 Memory Footprint and Feasibility

We discuss the memory footprint of Data Canopy in two scenarios: (1) when it is built with the optimal in-memory chunk size (256 bytes for our experimentation system) and (2) when, under memory pressure, it operates in Phase 2 of the out-of-memory policy (the chunk size grows to 64KB). These two scenarios correspond to the maximum and the minimum memory footprint of Data Canopy respectively. The experiment is on 100 columns and 40 million rows. Each node in Data Canopy is 8B. The analysis is conducted with the U workload and Data Canopy operates in online mode.

Figure 17 shows both the maximum memory footprint of Data Canopy in each scenario and the memory footprint after executing 2000 queries. We report the memory footprint of both univariate and bivariate statistics. In the case of univariate statistics, the maximum memory footprint is 1GB, and under memory pressure, it can incrementally shrink down to just 10MB. The maximum memory footprint of bivariate statistics is 32GB and, in a similar fashion, can shrink down to just 490MB. More generally, Data Canopy is able to vary its overall size (by changing its chunk size) to fit within the available main memory. Overall, the usage of the U workload remains less than one-third of the maximum size. In Appendix G, we provide a study of the feasibility of bivariate statistics in Data Canopy.

Update Experiments. We evaluate how Data Canopy handles updates in Appendix H.

4. RELATED WORK

Here we position Data Canopy against related efforts and we discuss how it advances the state of the art.

Modern Data Systems and Statistics. Data systems provide support to compute different statistics in the form of aggregate operations such as AVG, CORR etc. [84]. Also, query optimizers estimate query cardinality by using histogram statistics [21]. Recent approaches employ statistics for data integration [24, 42], time series analysis [66, 86], and learning [40, 68].

Despite widespread use of statistics in data systems, a framework to synthesize and reuse various statistical measures during exploratory statistical analysis does not exist. Data Canopy introduces such a framework, which replaces ad hoc calculation of statistics and brings opportunities to efficiently synthesize statistics from basic aggregates; compute and cache these basic aggregates ahead of time, and employ them to accelerate exploratory statistical analysis. Statistics in Data Canopy, primarily computed for exploratory analysis, can also be used within the data system for other tasks such as query optimization and data integration.

Improving Statistics. The widespread use of statistics has led to research on calculating fast statistics on large data sets. Some re-

search directions reduce the amount of data touched to compute statistics while providing guarantees on the accuracy: Robust sampling techniques are applied to trade accuracy for performance [19, 20, 23, 36, 79] and techniques based on discrete Fourier transform approximate all-pair correlations for time series [60]. Other research directions present solutions to compute statistics at scale in distributed settings: Cumulon is an end-to-end system, which optimizes the cost of calculating statistics on the cloud [46]. Similarly, other research directions optimize the calculation of various statistical measures by properly partitioning data in distributed settings [10, 23].

All these approaches innovate on how statistics are computed. Therefore, these approaches are all compatible with Data Canopy: Data Canopy can adopt one or even multiple of these approaches for computing basic aggregates. For example, Data Canopy in distributed settings, can incorporate aforementioned partitioning techniques to ensure that relevant data is stored at local nodes. The primary advantage that combining Data Canopy with these approaches has is that Data Canopy synthesizes statistics from basic aggregates and reuses these basic aggregates. In the presence of workloads exhibiting high locality and repetition, this significantly reduces data movement.

Data Cubes. Data cubes, widely applied in mining data warehouses, store data aggregated across multiple dimensions [38, 62]. Operators like roll-up, slice, dice, drill-down, and pivot allow data scientists to summarize or further resolve information along any particular dimension in the data cube. Various techniques to improve data cube performance have been studied: Sampling and other approximation techniques are used to reduce both the time required to construct the data cube and answer queries from it [11, 54, 81]. Some approaches only partially materialize data cubes [30, 31, 82], whereas others present strategies to build them adaptively [13], and in parallel settings [22]. One line of work proposes a simplified and flexible version of the data cube concept in form of small aggregates [59]. Furthermore, recent research designs data cubes for exploratory data analysis: Some research directions visualize aggregates stored in data cubes [50], others use them for ranking [80] as well as for interactive exploration [65].

Data cubes do not support a wide range of statistical measures. Specifically, they have no support for multivariate statistics such as correlation, covariance, or linear regression. Also, data cubes come with a high preprocessing and memory cost that results from calculating and storing aggregates grouped by multiple dimensions. In contrast, Data Canopy is both light-weight and is able to reuse and synthesize an extendible set of statistics using a relatively small set of basic aggregates. Furthermore, slices obtained from data cubes in OLAP settings can be explored using Data Canopy. Once data scientists have developed an understanding of the data set, then they can construct more complicated OLAP structures or run more detailed analytics on features and subsets of data that they have identified to be of interest. This approach is more efficient compared to building heavy OLAP structures up front for exploratory statistical analysis.

Query Caching and Prefetching. Query result caching enables database systems to reuse results of past queries to speed up future queries [44]. Most relevant to Data Canopy are approaches that enable reuse across different ranges by breaking down queries and caching query results [27, 51]. Data Canopy is inspired from these approaches and takes a step further: In addition to decomposing ranges, Data Canopy decomposes statistical measures into a set of basic aggregates that can be reused between them. As such, Data Canopy can synthesize descriptive and dependence statistics directly from this library of basic aggregates.

More recently, different approaches prefetch both data and query results to accelerate the process of data exploration. Forecache breaks the data down into regions called tiles, and prefetches them based on a data scientist's exploration signature [12]. Similar caching and prefetching strategies have been proposed for the process of data visualization [57]. Data Canopy advances this direction of work by providing a smart cache framework that can compute and maintain a library of basic aggregates that can be used as building blocks for a variety of statistical measures and machine learning algorithms.

Incremental Stream Processing. Similarly, in streaming scenarios incremental query processing decomposes data streams into smaller chunks and runs queries on these chunks: Window-based approaches partition data and queries such that future windows can make use of past computation [17, 18, 35, 56]. Certain approaches present strategies to incrementally monitor time series data [86] as well as update materialized views [15, 39]. Data Canopy is inspired from these approaches, and is readily applicable in streaming settings as it can be constructed in a single pass over the data set. When processing huge streams with limited memory, Data Canopy can function as a synopsis for answering a configurable set of statistical queries for exploratory statistical analysis. This synopsis can be constructed and updated incrementally.

In Appendix I we also discuss how Data Canopy relates to modern data exploration efforts.

5. CONCLUSION

We present Data Canopy a smart cache framework to accelerate the computation of statistics. Data Canopy breaks statistics down to their basic primitives, it caches and maintains those primitives, and uses them to synthesize future computations of (the same or different) statistics on the same or overlapping data. Contrary to state-of-the-art systems that need to always scan the whole data set to compute statistical measures, Data Canopy can interactively compute statistical measures without repeatedly touching the data; a property that becomes ever more important as data grows. Data Canopy can be computed both offline and online to speed up queries that overlap on data and on statistical measures. We demonstrate that Data Canopy brings significant speedup to exploratory statistical analysis and machine learning algorithms. This speedup continues to hold as the size of the data and the complexity of the exploration scenario (i.e. the number of repeated queries required to find the desired pattern) increases.

Acknowledgements. We thank the reviewers and Johannes Gehrke for their valuable feedback.

6. REFERENCES

[1] NumPy. http://www.numpy.org, 2013.
[2] Psycopg. http://initd.org/psycopg/, 2014.
[3] National Centers for Environmental Information (NCEI). https://www.ncei.noaa.gov, 2016.
[4] Kaggle Datasets. https://www.kaggle.com/datasets, 2017.
[5] The General Social Survey Datasets. http://gss.norc.org/Get-The-Data, 2017.
[6] The Quality of Government Institute Datasets. http://qog.pol.gu.se/data, 2017.
[7] Wofram – Descriptive Statistics. https://reference.wolfram.com/language/tutorial/DescriptiveStatistics.html, 2017.
[8] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful Query Specification with DataPlay. *Proceedings of the VLDB Endowment*, 5(12):1938–1941, 2012.
[9] 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 *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[10] F. Alvanaki and S. Michel. Tracking set correlations at large scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1507–1518, 2014.

[11] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting Approximations in Multidimensional Databases. *ACM SIGMOD Record*, 26(3):12–17, 1997.

[12] L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1363–1375, 2016.

[13] K. Beyer and R. Ramakrishnan. Bottom-up Computation of Sparse and Iceberg CUBE. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999.

[14] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[15] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

[16] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, 1999.

[17] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[18] S. Chandrasekaran, M. A. Shah, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, and F. Reiss. TelegraphCQ: continuous dataflow processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003.

[19] S. Chaudhuri, G. Das, and U. Srivastava. Effective Use of Block-Level Sampling in Statistics Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 287–298, 2004.

[20] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random Sampling for Histogram Construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 436–447, 1998.

[21] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

[22] Y. Chen, A. Rau-Chaplin, F. Dehne, T. Eavis, D. Green, and E. Sithirasenan. cgmOLAP: Efficient Parallel Generation and Querying of Terabyte Size ROLAP Data Cubes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 164, 2006.

[23] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[24] N. N. Dalvi and D. Suciu. Answering Queries from Statistics and Probabilistic Views. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 805–816, 2005.

[25] W. W. Daniel. Biostatistics: a foundation for analysis in the health sciences. *New York*, 1987.

[26] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[27] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching Multidimensional Queries Using Chunks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1998.

[28] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2014.

[29] M. Drosou and E. Pitoura. YmalDB: A Result-Driven Recommendation System for Databases. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 725–728, 2013.

[30] C. E. Dyreson. Information Retrieval from an Incomplete Data Cube. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 532–543, 1996.

[31] Y. Feng, D. Agrawal, A. El Abbadi, and A. Metwally. Range cube: efficient cube computation by exploiting data correlation. In *Proceedings. 20th International Conference on Data Engineering*, pages 658–669, 2004.

[32] M. Finnemore. Constructing statistics for global governance.

[33] S. Foundation. Sloan digital sky survey.

[34] T. R. Foundation. The R Project for Statistical Computing. https://www.r-project.org, 2016.

[35] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):57–72, 2007.

[36] P. B. Gibbons, Y. Matias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 466–475, 1997.

[37] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, and T. Suel. AQUA: System and Techniques for Approximate Query Answering. *Bell Labs - Technical Report*, 1998.

[38] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[39] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, 1995.

[40] Z. Guan, J. Wu, Q. Zhang, A. K. Singh, and X. Yan. Assessing and ranking structural correlations in graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 937–948, 2011.

[41] P. J. Guo. *Software Tools to Facilitate Research Programming*. PhD thesis, Stanford University, 2012.

[42] A. Y. Halevy. Structures, Semantics and Statistics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 4–6, 2004.

[43] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 721, 2006.

[44] J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 423–434, 1996.

[45] B. Howe, F. Ribalet, D. Halperin, S. Chitnis, and E. V. Armbrust. Sqlshare: Scientific workflow via relational view sharing.

[46] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2013.

[47] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, pages 277–281, 2015.

[48] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–293, 2016.

[49] A. E. W. Johnson, T. J. Pollard, L. Shen, L.-w. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. Anthony Celi, and R. G. Mark. Mimic-iii, a freely accessible critical care database. *Scientific Data*, 3:160035 EP –, 05 2016.

[50] M. Kahng, D. Fang, and D. H. P. Chau. Visual Exploration of Machine Learning Results Using Data Cube Analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*, pages 1:1—-1:6, 2016.

[51] A. M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal*, 5(1):35–47, 1996.

[52] A. Key, B. Howe, D. Perry, and C. R. Aragon. VizDeck: self-organizing dashboards for visual analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 681–684, 2012.

[53] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for snps with cloud computing. *Genome Biology*, 10(11), 2009.

[54] X. Li, J. Han, Z. Yin, J.-G. Lee, and Y. Sun. Sampling Cube: A Framework for Statistical OLAP over Sampling Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 779–790, 2008.

[55] E. Liarou and S. Idreos. dbTouch in action database kernels for touch-based data exploration. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1262–1265, 2014.

[56] E. Liarou, S. Idreos, S. Manegold, and M. Kersten. Enhanced stream processing in a DBMS kernel. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 501–512, 2013.

[57] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3):421–430, 2013.

[58] D. Madigan and R. Wasserstein. Statistics and science. *London Workshop on the Future of the Statistical Sciences*, 2013.

[59] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[60] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, 2010.

[61] H. Mühleisen and T. Lumley. Best of Both Worlds: Relational Databases and Statistics. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 32:1—-32:4, 2013.

[62] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, 1997.

[63] A. Nandi. Querying Without Keyboards. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[64] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. pages 2–4, 2005.

[65] S. Sarawagi and G. Sathe. I3: Intelligent, Interactive Investigation of OLAP Data Cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 589—-, 2000.

[66] S. Sathe and K. Aberer. AFFINITY: Efficiently Querying Statistical Measures on Time-Series Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 841–852, 2013.

[67] J. B. Saxe and J. L. Bentley. Transforming Static Data Structures to Dynamic Structures. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 148–168, 1979.

[68] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, New York, NY, USA, 2016. ACM.

[69] SciDB. SciDB-Py. *http://scidb-py.readthedocs.io/en/stable/*, 2016.

[70] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering Queries Based on Example Tuples. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 2014.

[71] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[72] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015.

[73] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(1):52–65, 2002.

[74] M. Stonebraker and J. Kalash. TIMBER: A Sophisticated Relation Browser (Invited Paper). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10, 1982.

[75] C. Surajit. Data Exploration Challenges in the Age of Big Data. In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*, 2016.

[76] A. Wasay, M. Athanassoulis, and S. Idreos. Queriosity: Automated Data Exploration. In *Proceedings of the IEEE International Congress on Big Data*, pages 716–719, 2015.

[77] M. L. Williams, K. M. Fischer, J. T. Freymueller, B. Tipoff, and A. M. TrÃl'hu. An earthscope science plan 2010-2020, feb 2010.

[78] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems. *Proceedings of the VLDB Endowment*, 7(10):903–906, 2014.

[79] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous Sampling for Online Aggregation Over Multiple Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 651–662, 2010.

[80] T. Wu, D. Xin, and J. Han. ARCube: Supporting Ranking Aggregate Queries in Partially Materialized Data Cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–92, 2008.

[81] X. Xie, X. Hao, T. B. Pedersen, P. Jin, and J. Chen. OLAP Over Probabilistic Data Cubes I: Aggregating, Materializing, and Querying. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 799–810, 2016.

[82] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing Iceberg Cubes by Top-down and Bottom-up Integration. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 2003.

[83] J. X. Yu, L. Qin, and L. Chang. Keyword Search in Relational Databases: A Survey. *IEEE Data Engineering Bulletin*, 33(1):67–78, 2010.

[84] Y. Zhao, P. Deshpande, and J. F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170, 1997.

[85] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *International workshop on Peer-To-Peer Systems, IPTPS 2006, Santa Barbara, CA, USA, February 27-28, 2006*.

[86] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 358–369, 2002.

APPENDIX

A. DATA CANOPY FOOTPRINT

As mentioned in Section 2.7, the Data Canopy footprint ($\mathscr{F}$) quantifies the memory overhead of Data Canopy. We define the Data Canopy footprint with respect to both a single statistic and a set of statistics, as the number of basic aggregates per column required to synthesize all instances of that statistic or set of statistics from Data Canopy. Below we elaborate how it applies to univariate and bivariate statistics.

Univariate Statistics. The Data Canopy footprint of univariate statistics is independent of the number of columns c. This is because to compute univariate statistics on a column, we require no information from other columns. For example, the Data Canopy footprint of mean is 1 because we need to keep only the sum for every column to synthesize the mean. Similarly the Data Canopy footprint of variance and standard deviation is 2.

Bivariate Statistics. The Data Canopy footprint of bivariate statistics depends on the number of columns c as they require information from pairs of columns. For example, to synthesize all pairwise correlations, we need sums and sums of squares of all c columns as well as $\frac{c\cdot(c-1)}{2}$ sums of pairwise products i.e., a total of $\frac{2+(c-1)}{2}$ basic aggregates per column.

Set of Statistics. The Data Canopy footprint is similarly defined for a set of statistics. For example, the Data Canopy footprint of mean and variance is 2 whereas the Canopy footprint of standard deviation, mean, and correlation is $\frac{2+(c-1)}{2}$.

Using terms from Table 2, we define the size of Data Canopy storing a set of statistics S as follows:

$$|DC(S)| = c \cdot (2 \cdot h - 1) \cdot \mathscr{F}(S) \cdot v_{st}$$

B. COMPOSABILITY

Here we define the concept of composability, which can be used to characterize the reusability of the basic aggregates cached by Data Canopy. Composability is the extent to which basic aggregates are shared by the set of statistics S supported by Data Canopy. Formally, it is the ratio between the number of basic aggregates shared by all members of S and the total number of basic aggregates required to synthesize S.

Let $\mathscr{B}(S)$ be the set of basic aggregates required to synthesize a statistic S, then the composability of S, given by $\mathscr{C}(S)$ is:

$$\mathscr{C}(S) = \frac{\bigcap_{i=1}^{i=|S|} \mathscr{B}(S_i)}{\bigcup_{i=1}^{i=|S|} \mathscr{B}(S_i)}$$

For instance, the composability of S = {mean, variance, standard deviation} is one-half. $\mathscr{C}(S)$ is zero when none of the statistics share any of the basic aggregates. On the other hand, $\mathscr{C}(S)$ is one when the same set of basic aggregates can be used to compute every member of S. A highly composable set of statistics will result in better reusability and lower memory requirement.

C. REAL WORLD DATA SETS

We quantify the number of columns and rows in publicly available data sets in healthcare, social science, and data science. We use these properties of real world data sets to design our experiments, and establish the feasibility of Data Canopy.

For healthcare, we look at MIMIC, a database with information about patients admitted to critical care units [49]. For quantitative social science, we look at data sets from the General Social Survey (GSS) [5] and the Quality of Government (QOG) Institute [6]. For data science, we look at the ten most frequently analyzed datasets on Kaggle, an online platform that hosts data science competitions [4]. All these data sets are both widely used and cited in their respective fields. Table A shows the number of columns and rows (range) for each of the data sets. Alongside, we provide the size of an optimal in-memory Data Canopy for each of the data sets.

Data sets	Columns (c)	Rows (r)	max |DC| (GB)
MIMIC	4 - 27	134 - 33M	0.9
QOG	44 - 2500	75 - 13885	2.9
GSS	10 - 296	100 - 10M	30.7
Kaggle	38 - 211	1.3M - 1.35M	2.1

Table A: Typical data sizes for data sets from healthcare, social science, and data science.

D. SCALABILITY

Here we provide additional experiments for scalability. In particular, we show how Data Canopy scales with hardware contexts and the number of queries.

Scaling with HW contexts. We first show the construction of Data Canopy scales as we increase the number of cores. We construct a complete Data Canopy (on 40 million rows and 100 columns) as we increase the amount of cores. Figure A shows that the construction time of Data Canopy goes down linearly with the number of cores. This is because the basic aggregates can be computed and cached completely in parallel.

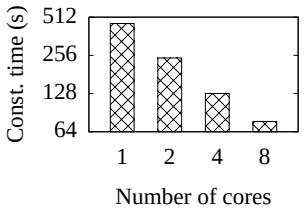


Figure A: The construction of Data Canopy scales linearly with the number of cores.

Scaling with the Number of Queries. We now show how Data Canopy scales when we increase the number of queries. We keep the same overall setting as before. We report scaling results only for the range-uniform workloads (U and Z). This is because for the range-zoom-in workloads (U_+ and Z_+), Data Canopy is completely built after the first 500 queries. Thus, all future queries are synthesized directly from Data Canopy (with minor data accesses to compute residual ranges), and the average response time remains constant thereafter.

Figure B shows the results. To make it easier to interpret, we report the average response time for every sequence of $50K$ queries. The more queries are processed, the more Data Canopy improves. For example, the last query takes up to 190.9× less time to compute than the first one. Toward the second half of the query sequence, the pace of improvement decreases as more queries can be synthesized directly from the library of basic aggregates without accessing the base data. The initial improvement in average response time is higher for workload Z as compared to workload U because queries exhibit more locality in the first one; once the library of basic aggregates is constructed, though, performance is nearly the same for both workloads as all queries are resolved directly from this library with only minor access to base data (for residual chunks).

E. MODEL VERIFICATION

We now verify the query cost model that we developed in Section 2.7. Similar to the analysis in Figure 8, we vary the chunk size for various number of rows and observe how this affects performance. The results are shown in Figure C. We report the total execution time of 10K queries from the U workload on 100 columns.

There are two observations. First, the experimental results verify the behavior we see from the model in Figure 8. That is, there is a convex shape and for all data sizes there is a common chunk size area where we get the optimal overall performance. Second, this area is actually quite large (the x-axis is logarithmic) and so picking any chunk size that is close enough to the center of this area gives optimal behavior. A positive side-effect of this is that we do not have to make our query cost model any more complex, i.e., by adding separate weights for when a cache access is a miss or a hit to capture the different latencies (traversing a segment tree will typically cause cache misses while scanning chunks at the endpoints of the query range (residual range) will typically cause a cache miss followed by more than one cache hits). Capturing simply the number of accessed cache lines allows us to get an estimate close enough in the optimal range, i.e., our analysis (as shown in Figure 8) estimates the optimal chunk size to be 220 bytes while Figure C shows that indeed 220 bytes is within the optimal range. An important side-effect of taking advantage of this behavior is that we do not need a training process for different machines (e.g., to figure out the cost of different accesses) - all we need is the cache line size. To fully optimize performance we pick a chunk size that is a multiple of the cache line. That is, the model gives us an optimal chunk size of 220 bytes, which we translate to a default chunk size of 256 bytes (4 times 64 which is the cache line size).

F. HANDLING MEMORY PRESSURE

Here we continue our analysis for the scenario when there is increased memory pressure. Specifically, we show how Data Canopy compares with Statsys (our baseline system that shares the codebase with Data Canopy but always compute statistics from data instead of basic aggregates). We set up an experiment with 8GB of main memory and Data Canopy operates in the online mode. The number of columns is fixed to 100 and we vary the number of rows to test the performance of Data Canopy across different stages of Phase 1 and 2 of out-of-memory policy.

Figure D shows the total execution time of 10K queries from the U workload under different memory pressures. In Phase 1, Data Canopy remains consistently 4× faster than Statsys. As the memory pressure builds up and Data Canopy transition to Phase 2, it continues to give a performance improvement of 4× even when only 50 percent of the data fits in memory. Under extreme memory pressure (only 25 percent of the data fits in memory) Data Canopy still results in 2× performance improvement.

G. BIVARIATE STATISTICS FEASIBILITY

Next, we show that under memory pressure Data Canopy can still efficiently support tens of thousands of bivariate statistics over a wide range of data sizes. In this analysis, the main memory budget is set to 16GB, and Data Canopy operates in Phase 2 of the out-of-memory policy. The chunk size is equal to a page size (64KB). All univariate segment trees are in memory. Figure E shows the number of bivariate segment trees that Data Canopy supports in the remaining amount of main memory across a wide range of data sizes. Each of the segment trees can be used to answer a bivariate statistic over any range of a pair of columns.

We observe that even for large data sets (1T rows and 1000 columns, data size to memory ratio of 1:250), Data Canopy can still efficiently support up to 10000 bivariate statistics, in addition to all univariate statistics.

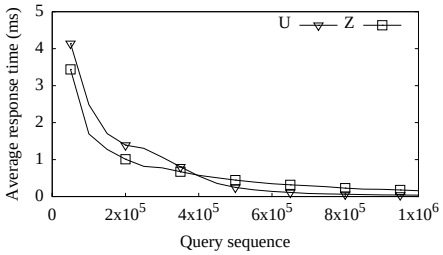


Figure B: As we increase the number of queries, the query response time continuously goes down (up to 190×).

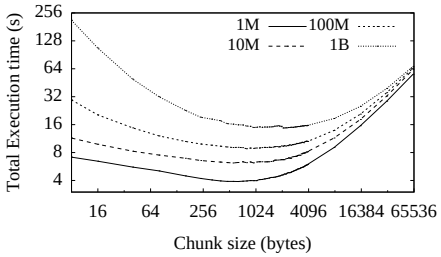


Figure C: The query performance of Data Canopy is a convex function of the chunk size.

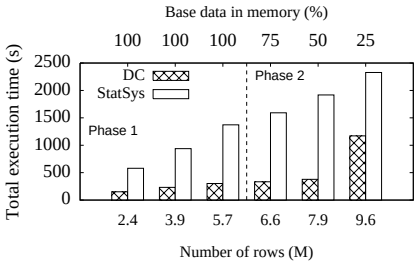


Figure D: In memory-constrained settings, Data Canopy provides 4× performance improvement over Statsys.

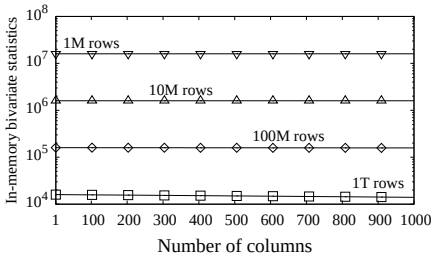


Figure E: Data Canopy can support tens of thousands of bivariate statistics.

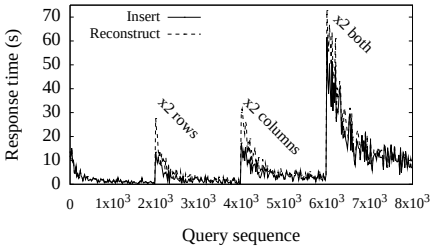


Figure F: Data Canopy gracefully handles new data.

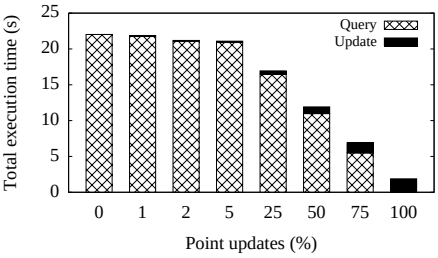


Figure G: Updates in Data Canopy result in negligible overhead.

H. UPDATES

Now we show that Data Canopy seamlessly handles updates.

Insertions. First, we show that Data Canopy efficiently handles insertions of new rows and new columns. We compare how Data Canopy incrementally handles updates to a strategy, where Data Canopy is built anew every time new data is added. We call this the reconstruct strategy. In this experiment, Data Canopy starts off with 25 columns and 100 million rows and operates in the online mode. New data is added in three phases: (1) the number of rows are doubled, (2) the number of columns are doubled, and (3) both the number of rows and columns are doubled. There is an interval of 2000 queries between each of the phases. At any point in time, we run the *U* workload that targets all data that is in the system. We report the response time as the sequence of queries passes through the three phases in Figure F. We observe that as new data is added, there is an initial increase in response time that converges to the optimal for both strategies. The incremental strategy employed by Data Canopy results in lower initial overhead as well as converges faster to stable performance as compared to the reconstruct strategy. This is because in the incremental strategy, both the insertion of new rows and new columns is handled in a lightweight manner (merely adding metadata to the catalog) and basic aggregates are materialized only when and if queries target the new data. In addition to this, the existing library of basic aggregates is completely reused, whereas with the reconstruct strategy, the library is built from scratch after every insertion phase.

Updates. Next, we show that response time is minimally impacted in the presence of updates to existing data. We show this for a varying percentage of updates in the workload. We set up an experiment with 100 columns and 100 million rows. We run 2000 queries from the *U* workload with varying percent of point updates in the workload. Figure G shows the total execution time as we increase the proportion (percentage) of point updates in the workload. As we increase the proportion of point updates in the workload, the number of read queries decreases resulting in an overall decrease in execution time. Throughout this time, the overhead introduced by point updates remains low. In low updates scenarios (1 to 5 percent point updates) the overhead is less than 1 percent. For extremely high update scenarios (25 to 75 percent point updates), the average update overhead is still below 10 percent of the total execution time.

I. DATA EXPLORATION

Here we discuss how Data Canopy relates to modern data exploration efforts. Data Exploration has received a lot of research interest within the data systems community [47, 76]. *Exploratory Interfaces* steer data scientists through the data space by providing both insights and further queries: Recent approaches discover relevant data objects based on relevance-feedback [28] or by performing a variation of faceted search [29]; Query recommendation systems help data scientists ask relevant questions based on the data set and their past interests [8, 70, 83]. *Visual Analytics* reduce the cognitive effort of data exploration by augmenting data systems with visual and gestural interfaces: Various approaches enable data scientists to visually browse data sets [73, 74, 78]; Recommendation systems automatically select an appropriate visualization given a data set [52]; DbTouch [55] and GestureDB [63] develop database kernels and languages that can be controlled by fingertips; recent efforts also work toward novel visualization languages [43]. *Approximate query processing* provides estimated answers to exploratory queries in orders of magnitude less time, by touching a fraction of base data. It uses samples of the data set to answer queries satisfying a user-defined accuracy [9, 37, 71].

Data Canopy as a framework for exploratory statistical analysis is complementary to all aforementioned efforts. None of the works described above are about making the process of computing statistics more interactive. Data Canopy can help make any process that contains iterative computation of statistics more interactive. Similarly, recommendation systems can make use of various descriptive and dependence statistics for faster and more informed recommendations. Data Canopy can also benefit from many of these research directions in the general field of data exploration. For instance, sampling and approximation techniques can be applied to create Data Canopy with approximate guarantees.