Optimally Leveraging Density and Locality for Exploratory Browsing and Sampling

Albert Kim*
MIT

Liqi Xu*
University of Illinois (UIUC)

Tarique Siddiqui
University of Illinois (UIUC)

Silu Huang
University of Illinois (UIUC)

Samuel Madden
MIT

Aditya Parameswaran
University of Illinois (UIUC)

ABSTRACT

Exploratory data analysis often involves repeatedly browsing a small sample of records that satisfy certain predicates. We propose a fast query evaluation engine, called NeedleTail, aimed at letting analysts browse a subset of the query result on large datasets as quickly as possible, independent of the overall size of the result. NeedleTail introduces DensityMaps, a lightweight in-memory indexing structure, and a set of efficient and theoretically sound algorithms to quickly locate promising blocks, trading off locality and density. In settings where the samples are used to compute aggregates, we extend techniques from survey sampling to mitigate the bias in our samples. Our experimental results demonstrate that NeedleTail returns results 7× faster on average on HDDs while occupying up to 23× less memory than existing techniques.

ACM Reference Format:
Albert Kim*, Liqi Xu*, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya Parameswaran. 2018. Optimally Leveraging Density and Locality for Exploratory Browsing and Sampling. In *HILDA'18: Workshop on Human-In-the-Loop Data Analytics, June 10, 2018, Houston, TX, USA.* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3209900.3209903

1 INTRODUCTION

When performing exploratory data analysis on new or unfamiliar datasets, analysts often issue queries, examine a small subset or sample of records, incrementally change their queries based on observations from this sample, and then repeat this process until they are satisfied. Thus exploratory data analysis has three facets that differentiate it from traditional data analysis: *(i) interactivity*, i.e., analysts wait to look at the results and cannot tolerate long delays [28], *(ii) incompleteness*, i.e., instead of looking at the millions of result records, analysts are satisfied with examining a small number (a few "screenfuls") of records, and *(iii) ad-hoc predicates*, i.e., analysts iteratively issue queries with small unpredictable modifications on the previous predicates, as well as periodically "zooming out" to explore different characteristics of a given data set.

When exploring data by repeatedly issuing ad-hoc queries, there are two primary ways users consume the results of the queries —*summarization* or *browsing* [18, 38]. That is, they either compute some aggregate summary statistics and then possibly visualize these statistics (summarization), or they examine a few tuples in the query result (browsing). The *browsing* or any-k use case is as common as *summarization*. For example, a recent study [31] reports that analysts often want to browse a subset of records to examine their analysis results. Moreover, many popular SQL IDEs [3–5] implicitly limit the number of result records displayed, recognizing the fact that users often do not need to, nor do they have the time to see all of the result records. Even outside of the context of SQL IDEs, in tabular interfaces such as Microsoft Excel or Tableau's Table View, users only browse or examine a subset of records—depending on the user, this number could be in the *hundreds or thousands, even though the result set can number in the millions or billions*

Perhaps surprisingly, despite the wealth of work on exploratory data analysis, there is little work that has addressed this *browsing* problem: *how do we quickly return a small subset of records that satisfy arbitrary user-specified predicates*, without requiring the records be the "top" ones (for some definition of top), or requiring that they be a random subset of all satisfying records. We call this any-k problem.

Current database techniques are not optimal for any-k problem, in terms of both query performance and memory consumption. For example, existing databases support LIMIT/any-k by returning the first k result records as soon as they are ready, via the selection of query plans that can pipeline and produce early results. This is often not interactive, especially if the query involves selective predicates. Similarly, traditional indexing structures, such as B+Trees, could efficiently answer some any-k queries. However, to support any-k queries with an arbitrary combination of predicates, as is typical in exploratory browsing, we would need B+Trees on every single attribute or combination of attributes, which will be prohibitive in terms of space and maintenance costs [9, 26, 33, 40]. Bitmap indexes are a more space-efficient approach and can support arbitrary predicates via bitwise operations. But even so, storing a bitmap in memory for every single value for every single attribute (e.g., 10s of values for 100s of attributes) is impossible for large datasets.

To address these limitations, we introduce a new data exploration engine, NeedleTail.[1] Specifically, we develop indexing and query evaluation techniques to address the any-k problem. We now describe these contributions and the underlying challenges.

DensityMaps**: A Lightweight Indexing Scheme.** Inspired by bitmap indexes, we develop a simple and lightweight indexing scheme called DensityMaps. Unlike bitmaps, which store a column of bits for each distinct value of an attribute, DensityMaps store an array of densities for each *block* of records, where the array contains one entry per distinct value of the attribute. This representation

[1] We named NeedleTail after the world's fastest bird [6].

HILDA'18, June 10, 2018, Houston, TX, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/3209900.3209903

allows DensityMaps to be 3-4 orders of magnitude smaller than bitmaps, so we can easily store a density map for every attribute in the data set, and these can comfortably fit in memory even for large datasets. Note that prior work has identified other lightweight statistics that help rule out blocks that are not relevant for certain queries [10, 13, 25, 30, 39]; we compare against the appropriate ones in our experiments in Section 7.

DensityMaps: **Density vs. Locality.** In order to use DensityMaps to retrieve k records that satisfy query constraints, one approach would be to identify blocks that are likely to be "dense" in that they contain more records that satisfy the conditions and preferentially retrieve those blocks. However, this *density-based* approach may lead to excessive random access. Because random accesses—whether in memory, on flash, or on rotating disks—are generally orders of magnitude slower than sequential accesses, this is not desirable. Another approach would be to identify a sequence of consecutive blocks with k records that satisfy the conditions, and take advantage of the fact that sequential access is faster than random access, exploiting *locality*. In this paper, we develop algorithms that are optimal from the perspective of density and locality respectively.

Overall, while we would like to optimize for both density and locality, optimizing for one usually comes at the cost of the other, so developing the globally optimal strategy to retrieve k records is non-trivial. We develop a simple cost model for storage media in this paper, and use this to develop an algorithm that is optimal from the perspective of overall I/O . We further extend the density and locality-optimal algorithms to develop a hybrid algorithm that fuses the benefits of both approaches. We integrated all four of these algorithms, coupled with indexing structures, into our NeedleTail. On both synthetic and real datasets, we observe that NeedleTail *can be several orders of magnitude faster than existing approaches* when returning k records that satisfy user conditions.

Aggregate Estimation with Any-k Results. In some cases, instead of just optimizing for retrieving any-k, it may be important to use the retrieved results to estimate some aggregate value. Although NeedleTail can retrieve more samples than random sampling in a given time period, its estimate of the aggregate value may be biased, since NeedleTail may preferentially sample from certain blocks. This is especially true if the attribute being aggregated is correlated with the layout of data on disk or in memory. We employ *survey sampling* [20, 29] techniques to support accurate aggregate estimation while retrieving any-k records. In particular, we employ *cluster sampling* techniques to reason about block-level sampling, along with *unequal probability estimation* techniques to correct for the bias. With these changes, NeedleTail is able to achieve error rates similar to pure random sampling—our gold standard—in *much less time*, while *returning multiple orders of magnitude more records* for the analyst to browse. Thus, even when computing aggregates, NeedleTail is substantially better than other schemes.

Outline. The chief contribution of this paper is the characterization of the any-k problem (Section 2) and the development of NeedleTail, an efficient data exploration engine for both browsing and aggregate estimation that retrieves samples orders of magnitude faster than other approaches. NeedleTail's design includes its indexing structure (Section 3), retrieval algorithms (Section 4), and statistical techniques to correct for biased sampling in aggregate estimation (Section 5), with extensions for complex queries (Section 6). Figure 1 depicts the overall architecture of NeedleTail.

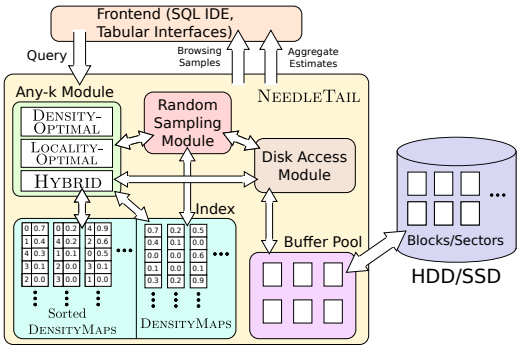


Figure 1: NeedleTail **Architecture**

2 PROBLEM FORMULATION

We now formally define the any-k problem. We consider a standard OLAP data exploration setting where we have a database D with a star schema consisting of continuous measure attributes M and categorical dimension attributes A. For simplicity, we focus on a single database table T, with r dimension attributes and s measure attributes, leading to the schema: $T = \{A_1, A_2, ..., A_r, M_1, M_2, ..., M_s\}$; our techniques generalize beyond this case, as we will show later. We use δ_i to denote the number of distinct values for the dimension attribute A_i with distinct values $\{V_i^1, V_i^2, .., V_i^{\delta_i}\}$.

Consider a selection query Q on T where the selection condition is a boolean formula formed out of equality predicates on the dimension attributes A. We define the set of records which form the result set of query Q to be the *valid records* with respect to Q. As a concrete example, consider a data analyst exploring campaign finance data. Suppose they want to find any-k individuals who donated to Donald Trump, live in a certain county, and are married. Here, the query Q on T has a selection condition that is a conjunction of three predicates—donated to Trump, lives in a particular county, and is married.

We now define an any-k query Q_k as the query which returns k valid records out of the set of all valid records for a given query Q. Q_k can be written as follows:

SELECT **ANY-K**(∗) FROM T WHERE ⟨CONDITION⟩

For now we consider simple selection queries of the above form; we extend our approach to support aggregates in Section 5, grouping in Section 6 and joins in technical report [24].

We formally state the any-k sampling problem for simple selection queries as follows:

Problem 1 (Any-k Sampling). *Given an any-k query Q_k, the goal of any-k sampling is to retrieve any k valid records in as little time as possible.*

Unlike random sampling, any-k does not require the returned records to be randomly selected. Instead, any-k sampling prioritizes query execution time over randomness. We will revisit the issue of randomness in Section 5.

3 DENSITY MAP INDEX

To support the fast retrieval of any-k samples, we develop a lightweight indexing structure called the DensityMap. Our design starts from the observation that modern hard disk drive (HDDs) typically have 4KB minimum storage units called sectors, and systems may only read or write from HDDs in whole sectors. Therefore, it takes the same amount of time to retrieve a block of data as it retrieves a

single record. DensityMaps take advantage of this fact to reason about the data at the block-level, rather than at the record-level as bitmaps [11, 34, 41, 42] do. Similarly, SSD and RAM access pages or cache lines of data at a time.

Thus, for each block, a DensityMap stores the frequency of set bits in that block, termed the *density*, rather than enumerating the set bits. Formally, for each attribute A_i, and each attribute value V_i^j taken on by A_i, we store a DensityMap D_i^j, consisting of λ entries, one corresponding to each block on disk. We can express D_i^j as $\{d_{i_1}^j, d_{i_2}^j, .., d_{i_k}^j, .., d_{i_\lambda}^j\}$, where $d_{i_k}^j$ is the percentage of tuples in block k that satisfy the predicate $A_i = V_i^j$. While DensityMaps are stored per column, the underlying data is assumed to be stored in row-oriented fashion. Note that DensityMaps could equivalently be applied to a column-store setting.

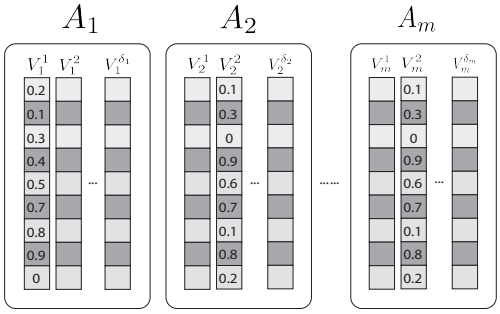


Figure 2: *DensityMaps*

Example 1. *The table in Figure 2 is stored over 9 blocks. The density map D_1^1 for V_1^1 is $\{0.2, 0.1, 0.3, 0.4, 0.5, 0.7, 0.8, 0.9, 0\}$, indicating 20 percent of tuples in block 1 and 10 percent of tuples in block 2 have value V_1^1 for attribute A_1 respectively.*

DensityMaps are a very flexible index structure as they can estimate the percentage of valid records for any ad-hoc query with single or nested selection constraints. For queries with more than one predicate, we can combine multiple DensityMaps together to calculate the estimated percentage of valid records per block, multiplying densities for conjunction and adding them for disjunction, akin to selectivity estimation in query optimization [14]. As in query optimization, this assumption may not always hold, but as we demonstrate in our experiments on real datasets, it still leads to effective results. Furthermore, DensityMaps drastically reduce the number of disk accesses by skipping blocks whose estimated densities are zero (and thus definitely do not contain valid records). Thus, compared to bitmaps, DensityMaps are a coarser statistical summary of valid records in each block for each attribute value.

4 ANY-K ALGORITHMS

In this section, we introduce four algorithms that take advantage of DensityMaps to perform fast any-k sampling, making use of the following two observations.

Observation 1 (Density: Denser is better). *Retrieving a block with high density is preferable to retrieving a low density block.*

First, a high density block has more valid records than a low density block. Thus, it is more beneficial to retrieve the high density block, so that overall, fewer blocks are retrieved.

Observation 2 (Locality: Closer is better). *Retrieving neighboring blocks is preferable to retrieving blocks which are far apart.*

In a HDD, the time taken to retrieve a block from disk can be split into *seek time* and *transfer time*. The seek time is the time it takes to locate the desired block, and the transfer time is the time required to actually transfer the data. Blocks which are far apart incur additional seek time, while neighboring blocks typically only require transfer time. Thus, retrieving neighboring blocks is preferred on HDDs. For in-memory data and SSDs, the locality observation may not be as important, since the random I/O performance of these storage media is not as poor as it is on HDDs. For our purposes, we focus on the HDDs, but we also evaluate our techniques on SSDs.

4.1 Two Extremes

Our two basic any-k sampling algorithms take advantage of each of these observations: Density-Optimal optimizes for density while Locality-Optimal optimizes for locality. These two algorithms are *optimal extremes*, favoring just one of locality or density. Details are provided in our technical report [24].

Density-Optimal **Algorithm.** The goal of Density-Optimal is to use our in-memory DensityMap index to retrieve the densest blocks until k valid records are found. To this end, we adapt the threshold algorithm (TA) by Fagin et al. [12] for top-k processing on sorted lists. The "scores" of the objects in TA are the densities of the blocks, and the aggregation functions in TA are a combination of products and summations based on the ANDs and ORs in the query predicate. To support this algorithm, we need to maintain sorted DensityMaps in addition to regular DensityMaps. One small difference is that the original threshold algorithm would attempt to find the p densest blocks. However, in our setting, we do not know the value of p in advance; we only know k, the number of valid tuples required, so we need to set the value of p dynamically.

Depending on the order of the blocks returned by Density-Optimal, the system may perform many unnecessary random I/O operations. For example, if Density-Optimal returns blocks $\{B_{100}, B_1, B_{83}, B_3\}$, the system may read block B_{100}, seek to block B_1, and then seek back to block B_{83}. Instead, we can sort the blocks $\{B_1, B_3, B_{83}, B_{100}\}$ before fetching them from disk, thereby minimizing random I/O and overall query execution time.

Locality-Optimal **Algorithm.** Locality-Optimal prioritizes for locality rather than density, aiming to identify the shortest sequence of blocks that guarantee k valid records. The naive approach to identify this would be to consider the sequence formed by every pair of blocks (along with all of the blocks in between)—leading to an algorithm that is quadratic in the number of blocks. Instead, Locality-Optimal is linear in the number of blocks. Locality-Optimal moves across the sequence of blocks using a sliding window formed using a start and an end pointer, and eventually returns the smallest possible window with k valid records.

4.2 Hybrid Any-k Algorithms

The two desired properties of density and locality can often be at odds with each other depending on the data layout; dense blocks may be far apart, and neighboring blocks may contain many blocks which have few valid records. In this subsection, we first present a cost model to estimate the I/O cost of an any-k algorithm.

A Simple I/O Cost Model. To set up the cost model for I/O for HDDs, we profile the storage system. We randomly choose various starting blocks and record the time taken to fetch other blocks

that are varying distances away. Our experiment demonstrates that with block size equal to 256KB, the I/O cost is smallest when doing a sequential I/O operation to fetch the next block (~ 2ms), and increases with the distance up to a certain maximum distance t after which it becomes constant (~ 12ms). Formally, for two blocks i and j, we model the cost of fetching block j after block i as:

$$RandIO(i,j) = \begin{cases} cost(i,j) & \text{if } |j-i| \leq t \\ constant & otherwise \end{cases}$$

On the other hand, the I/O cost model for SSDs is different from the one we see for HDDs. Overall, we see a constant time (~ 0.6ms) to fetch a block independent of the block distance

IO-Optimal **Algorithm.** IO-Optimal considers both density and locality to find the set of blocks with the minimum I/O cost overall. Given our cost model, we can use dynamic programming to find the optimal set of blocks with k valid records.

We define $C(s,i)$ as the minimal cost to retrieve s estimated valid records when block i is amongst the blocks fetched. We define $Opt(s,i)$ as the cost to retrieve the optimal set of blocks with s estimated valid records when considering the first i blocks. Finally, we denote s_i as the estimated number of valid records inside block i. With this notation, we have:

$$C(s,i) = \min \begin{cases} C(s-s_i, j) + RandIO(j,i), \quad \forall j \in [\,i-t, i-1] \\ Opt(s-s_i, i-t-1) + RandIO(i-t-1, i) \end{cases}$$

$$Opt(s,i) = \min \begin{cases} C(s,i) \\ Opt(s, i-1) \end{cases}$$

The intuition is as follows: for each block i that has s_i estimated valid records, either the block can be in the final optimal set or not. If we decide to include block i, the cost is the minimum cost amongst the following: *(i)* the smallest I/O cost of having $s - s_i$ samples at block j where $|i-j| \leq t$, plus the cost of jumping from block j to i (i.e., $C(s-s_i, j) + RandIO(j,i)$), or *(ii)* the optimal cost at block $i-t-1$, plus the random I/O cost of jumping from some block in the first $i-t-1$ blocks to block i (i.e., $Opt(s-s_i, i-t-1) + RandIO(i-t-1, i)$).

For the second expression, if we exclude block i, then the optimal cost is the same as the optimal cost at block $i-1$. Consequently, the optimal cost at block i is the smallest value in these two cases.

Hybrid **Algorithm.** Even though IO-Optimal is able to return the optimal I/O cost for fetching any-k samples, its much higher computation cost makes it impractical for large datasets. We propose Hybrid which simply selects between the best of Density-Optimal and Locality-Optimal, using our I/O cost model.

5 AGGREGATE ESTIMATION

So far, our any-k algorithms retrieve k records without any consideration of how representative they are of the entire population. If these records are used to estimate an aggregate, there could be bias in this value due to possible correlations between the value and the data layout. While this is fine for browsing, it leaves the user unable to make any statistically significant claims about the aggregated value. To address this problem, we make two simple adjustments to extend our any-k algorithms. First, we introduce a Two-Phase *sampling scheme* where we add small amounts of random data to our any-k estimates—this allows us to *calibrate and correct for* bias. Second, we correct the bias by leveraging techniques from survey sampling literature. Such approaches have been employed in other approximate query processing settings [21, 23, 27], but their application to any-k is new.

Two-Phase **Sampling.** We propose a Two-Phase sampling scheme, in which we collect a large proportion (1-α) of the k requested samples using an any-k algorithm, and collect the rest (α) in a random fashion. The user chooses the parameter α upfront based on how much random sampling they wish to add. While a larger α may reduce the number of total samples needed to obtain a statistically significant result, the time taken to retrieve random samples greatly exceeds the time taken to retrieve samples based on our any-k algorithms. Therefore, α needs to be carefully chosen; we experiment with α in Section 7. We provide a more formal description of our sampling procedure in our technical report [24].

Unequal Probability Estimation. Within the Two-Phase sampling scheme, the probability a block is sampled is not uniform. Therefore, we must use an *unequal probability estimator* [37] and inversely weigh samples based on their selection probabilities. We apply two different estimators for this: the Horvitz-Thompson [20] and ratio [29] estimators. Even though the Horvitz-Thompson estimator is an unbiased estimator, its variance is known to often be large. On the other hand, the ratio estimator is known to have low variance. We describe further details and formulae in [24].

6 GROUPING

Rather than just a simple any-k query, users may want to retrieve k values per group. Unlike a traditional GROUP-BY, which returns a single aggregated tuple per group, we require k sample non-aggregated tuples per group, so that users can aggregate the tuples per group in any way they wish. Although a trivial solution would be to run a separate any-k query per group, we propose an algorithm that can share the computation across groups in the common case when users want k values per group.

In order to run our any-k algorithms for all groups, we first define the *combined density* of the lth block as the multiplication of two factors: (1) the density of the lth block with respect to predicate S, and (2) the sum of the densities for group values in the grouping attribute, A_G, in the lth block that still need to be sampled. The first factor has been discussed previously, while the second factor can be defined as:

$$d^*_{G_l} = \frac{1}{\text{RPB}} \sum_{j=1}^{\delta_G} min\left(k - r^j_G\ ,\ d^j_{G_l} \times \text{RPB}\right) \tag{1}$$

where RPB is *records_per_block*, r^j_G is the number of samples already retrieved for the grouping attribute value V^j_G, and $d^j_{G_l}$ is the density of the lth block for the value V^j_G. The expression inside the *min* function estimates the number of expected records in block l for each group V^j_G, but limits the estimate by the number of samples left to be retrieved for that group. Thus, the combined density ($d_{S_l} d^*_{G_l}$), where d_{S_l} is the density of the lth block with respect to predicate S, gives priority to groups that have had fewer than k samples retrieved so far, and groups that already have k samples no longer contribute to the combined density. The 1/RPB in front of the summation for $d^*_{G_l}$ acts a normalization factor to ensure that $d^*_{G_l}$, and thereby $d_{S_l} d^*_{G_l}$, are both density values between 0 and 1.

With this combined density estimate $d_{S_l} d^*_{G_l}$, we can now construct an iterative any-k algorithm for grouped sampling operations:

(1) Update all densities using with $d_{S_l} d^*_{G_l}$. (2) Run one of the any-k algorithms to retrieve ψ blocks with the highest combined density. (3) Update the densities of the ψ blocks as 0. (4) If k samples still have not been retrieved for each group, go back to step (1). We present the full pseudocode, extensions to multi-attribute grouping and join, and the selection of ψ in the technical report [24].

7 PERFORMANCE EVALUATION

In this section, we evaluate NeedleTail, focusing on runtime, memory consumption, and accuracy of estimates. We show that our DensityMap-based any-k algorithms outperform any "first-to-k-samples" algorithms using traditional OLAP indexing structures such as bitmaps or compressed bitmaps on a variety of synthetic and real datasets. In addition, we empirically demonstrate that our Two-Phase sampling scheme is capable of achieving as accurate an aggregate estimation as random sampling in a fraction of the time. In our technical report [24], we study the impact of a number of parameters on NeedleTail including: *(i)* data size, *(ii)* number of predicates, *(iii)* density, *(iv)* block size, and *(v)* granularity. We also demonstrate that the join variant of any-k algorithms provide substantial speedups for key-foreign key joins.

7.1 Experimental Settings

We now describe our experimental workload, the evaluated algorithms, and the experimental setup.

Airline Dataset [1]. This dataset contained the details of all flights within the USA from 1987–2008, sorted based on time. It consisted of 123 million rows and 11 attributes with a total size of 11 GB. For our experiments on error (described later), we estimated the average arrival delay, average departure delay, and average elapsed time for flights. The details of the queries are listed in Tables 1.

Q#	Query Selectivity	Result Tuples	1% any-k	Equality Predicates on
Q1	0.004%	5499	54	carrier AND destination
Q2	0.007%	8459	84	month AND origin AND destination
Q3	0.033%	41000	410	month AND origin
Q4	0.785%	969300	9693	dayofwork AND origin
Q5	1.172%	1447400	14474	origin

Table 1: Query details for airline workload.

We describe experiments on another real dataset (Taxi) as well as a synthetic dataset in our technical report [24]; results are similar.

We compared our any-k algorithms presented in Section 4 against the following "first-to-k-samples" baselines: *(i)* Bitmap-Scan: Assuming we have bitmaps for every predicate, we use bitwise operations to construct a resultant bitmap corresponding to the valid records. We then retrieve the first k records whose bits are set in this bitmap. *(ii)* Lossy-Bitmap [39]: Lossy-Bitmap is a variant of bitmap indexes where a bit is set for each block instead of each record. For each attribute value, a set bit for a block indicates that at least one record in that block has that attribute value. During data retrieval, we perform bitwise operations then fetch k records from the first few blocks which have their bit set. *(iii)* EWAH: This baseline is identical to Bitmap-Scan, except the bitmaps are compressed using the Enhanced Word-Aligned Hybrid (EWAH) technique [2, 26].

We choose in-memory bitmap indexes as our baseline as opposed to other common indexing techniques, such as B+Trees, for multiple reasons: first, given the high storage cost and unknown exploratory workloads, it would be infeasible to store B+Tree indexes for every combination of attributes [26], thereby resulting in false-positives when using a B+Tree that "misses" some of the predicates in the query, and taking a lot more time to answer any-k queries; second, even if the B+tree index covering the query predicates is available, the I/O cost of retrieving the tuples would be identical to that of bitmap indexes (since they would retrieve tuples in the same order), while its computation cost is not likely to be smaller (since B+Tree traversals are branch-intensive and relatively inefficient on modern CPUs, compared to bitmap indexes that are parallelizable, and admit efficient bit manipulation operations). Thus, in-memory bitmap indexes end up being a suitable proxy for the *minimum possible time taken by traditional indexes*, including B+Trees.

Setup. All experiments were conducted on a 64-bit Linux server with 8GB of main memory and 1TB HDD. We ran 5 trials (30 trials for the random sampling experiments) for each query on each dataset. To minimize experimental variance, we discarded the trials with the maximum and minimum runtime and reported the average of the remaining.

7.2 Query Execution Time

Figures 3 shows the runtimes of our algorithms over 5 diverse queries for the airline dataset. We split the runtime of each query into I/O time (bottom part of each bar) and CPU time (top part of each bar). For each query and sampling rate, we normalized the runtime of each algorithm by the largest runtime across all algorithms, while also reporting actual runtime (in ms) taken by Hybrid and the maximum runtime.

In Figure 3, we noticed that our any-k algorithms consistently outperformed the bitmap-based baselines: Density-Optimal had a speedup of up to 8× compared to Bitmap-Scan and EWAH, while Locality-Optimal had a speedup of up to 7×. Across all queries, when sampling rate equals 1%, Density-Optimal and Locality-Optimal were on average 3× and 5× faster than Bitmap-Scan and EWAH, despite having a much smaller memory footprint. We also observed that, on HDDs, the I/O time was the bottleneck, occupying 90% of the runtime on average. Moreover, the I/O time of Density-Optimal and Locality-Optimal was on average 4× and 3× faster than I/O time of Bitmap-Scan and EWAH. Compared with Locality-Optimal, Density-Optimal fetched up to 10% less blocks, resulting in less I/O time and consequently query execution time than Locality-Optimal in all of cases. Hybrid ends up always selecting the faster algorithm in both this and the taxi workload, with an average speedup of 4×. For example, for Q4 with 1% sampling rate Hybrid's time is closer to Density-Optimal, and half of that of Locality-Optimal.

We also compared the runtimes of any-k algorithms with baselines on SSDs [24]. We observed that Density-Optimal was the fastest as it fetches the least number of blocks.

7.3 Memory Consumption

Dataset	Size	# Tuples	Card.	Bitmap	EWAH	Lossy	DensityMap
Synthetic	8GB	100M	16	191MB	183MB	0.06MB	4MB
Taxi	21GB	253M	64	1937MB	664MB	0.7MB	42MB
Airline	11GB	123M	805	11852MB	744MB	4.0MB	255MB

Table 2: Memory consumption of index structures.

Table 2 reports the amount of memory used by DensityMaps compared to the other three bitmap baselines on three datasets [24]. We observed that DensityMaps are very lightweight and consumed around 51×, 47×, and 47× less memory than uncompressed bitmaps

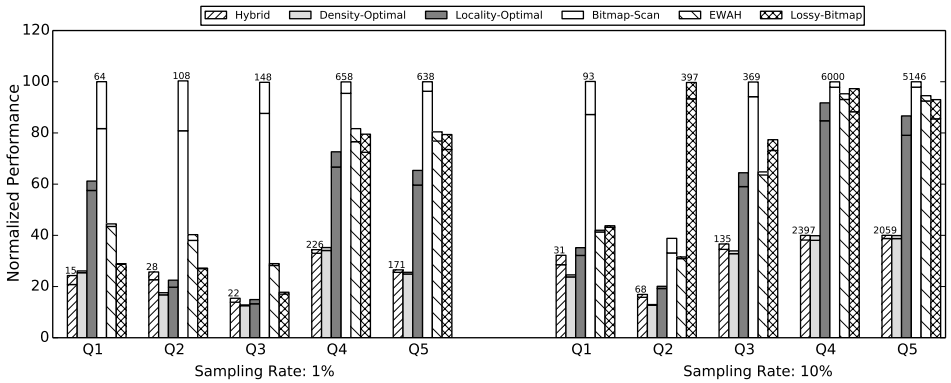
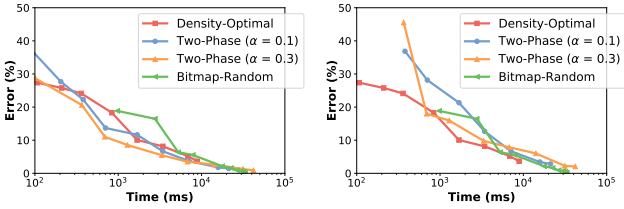


Figure 3: Query runtimes for airline workload on a HDD.



(a) Airline - Ratio Estimator **(b) Airline - HT Estimator**

Figure 4: Time vs empirical error.

respectively in the three datasets. Even with EWAH-compression, we observed an almost 49× reduction in size for the taxi dataset for DensityMaps relative to EWAH. In the airline dataset, since the selectivity of each attribute value is low, EWAH compressed the bitmaps much better than in the other two datasets. Still, EWAH consumed 3× more memory than DensityMap. Lastly, since Lossy-Bitmap requires only one bit per block while DensityMap is represented as a 64-bits double per block respectively, Lossy-Bitmap unsurprisingly consumed less memory than DensityMap. However, as we showed in Section 7.2, the smaller memory consumption incurred a large cost in query latency due to the large number of false positives (e.g., Q3 with sampling rate 10% in Figure 3), especially when the number of predicates is large and exhibit complex correlations. In comparison, the DensityMap-based any-k algorithms were orders of magnitude faster than the baselines, while still maintaining a modest memory footprint ($\sim$ 0.1% of original dataset).

7.4 Time vs Error Analysis

Using the Two-Phase sampling techniques in Section 5, we can obtain estimates of aggregate values on data; here we experiment with $\alpha = 0\%, 10\%, 30\%$ random samples, and use the Density-Optimal algorithm, since it ended up performing the most consistently well across queries and media. We compared these results with pure random sampling (Bitmap-Random) using bitmaps on a HDD. We used the same set of queries as in Tables 1. For each query, we varied the sampling rate and measured the runtime and the empirical error of the estimated aggregate with respect to the true average value. Figure 4 depicts the average results in log scale.

We find that Density-Optimal performs better than the Two-Phase sampling scheme with the ratio estimator for the initial period until about 100ms, after which the Two-Phase sampling schemes perform better than Density-Optimal and Bitmap-Scan. We found this behavior repeated across other queries and trials: Density-Optimal sometimes ends up having very low error (Figure 4a), and sometimes fairly high error [24], but the Two-Phase sampling schemes consistently achieve low error relative to Density-Optimal. This is because Density-Optimal's accuracy is highly dependent on the correlation between the data layout and the attribute of interest, and can sometimes lead to highly biased results. At the same time, the Two-Phase sampling schemes return much more samples and much more accurate estimates than Bitmap-Random, effectively supporting both browsing and sampling.

8 RELATED WORK

Data Skipping. A group of indexing techniques, including Lossy-Bitmap [39], SMA [30] and variants of SMA [10, 13, 25], were developed to track aggregate attribute information at the block level for query processing. However, as shown in Section 7, DensityMaps are significantly better-suited for the any-k problem. Like NeedleTail, Sun et al.'s data skipping technique [35, 36] skips blocks, via partitioning, but requires a workload up-front.

Approximate Query Processing. In the past decade, a number of approximate query processing techniques [15, 16, 22] and systems [7, 8] have emerged that allow users to trade off query accuracy for interactive response times, by employing random sampling. One related vein of work performs online sampling [17, 19, 22, 32], using techniques that are either similar to Bitmap-Random or Disk-Scan, in order to achieve adequate randomization. In contrast, NeedleTail primarily focuses on any-k sampling. This allows NeedleTail to avoid accessing data in random order, avoiding expensive upfront randomization or inefficient random access to data at runtime. Moreover, our Two-Phase sampling technique returns much larger samples than random sampling, but in much less time and with comparable accuracy.

Our technical report [24] provides a detailed description of related work.

9 CONCLUSIONS

We presented NeedleTail, a data exploration engine that by retrieving any-k valid records for arbitrary queries as quickly as possible. We proposed DensityMaps, a lightweight index structure, as well as four any-k sampling algorithms built on top of simple cost models. Our experimental evaluations demonstrated that NeedleTail is effectively able to trade-off density and locality to speed up query runtimes on average by 7× and is able to obtain similar error rates to random sampling in much less time.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grant IIS-1513407, IIS-1633755, IIS-1652750, and IIS-1733878 awarded by the National Science Foundation.

REFERENCES

[1] [n. d.]. Airline Dataset. ([n. d.]). http://stat-computing.org/dataexpo/2009/the-data.html [Online; accessed 30-Dec-2015].

[2] [n. d.]. C++ EWAH Implementation. https://github.com/lemire/EWAHBoolArray. ([n. d.]).

[3] [n. d.]. MySQL phpMyAdmin. ([n. d.]). http://docs.phpmyadmin.net/en/latest/config.html

[4] [n. d.]. Oracle SQL Developer. ([n. d.]). http://oracle.com/technetwork/developer-tools/sql-developer/

[5] [n. d.]. PostgreSQL phpMyAdmin. ([n. d.]). http://oracle.com/technetwork/developer-tools/sql-developer/

[6] [n. d.]. Swift. https://en.wikipedia.org/wiki/Needletail. ([n. d.]).

[7] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua Approximate Query Answering System *(SIGMOD)*. 574–576. https://doi.org/10.1145/304182.304581

[8] Sameer Agarwal et al. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.

[9] Manos Athanassoulis and Stratos Idreos. 2016. Design Tradeoffs of Data Access Methods. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2195–2200.

[10] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[11] Chee-Yong Chan and Yannis E Ioannidis. 1998. Bitmap index design and evaluation. In *ACM SIGMOD Record*, Vol. 27. ACM, 355–366.

[12] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66, 4 (2003).

[13] Phil Francisco et al. 2011. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbooks* 3 (2011).

[14] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.

[15] Minos N. Garofalakis and Phillip B. Gibbon. 2001. Approximate Query Processing: Taming the TeraBytes. In *VLDB*. 725–. http://dl.acm.org/citation.cfm?id=645927.672356

[16] Phillip B Gibbons. 2001. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*. 541–550.

[17] Peter J Haas and Joseph M Hellerstein. 1999. Ripple joins for online aggregation. *ACM SIGMOD Record* 28, 2 (1999), 287–298.

[18] Pat Hanrahan. 2012. Analytic database technologies for a new kind of user: the data enthusiast. In *SIGMOD*. ACM, 577–578.

[19] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD Conference*.

[20] Daniel G Horvitz and Donovan J Thompson. 1952. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association* 47, 260 (1952), 663–685.

[21] Ying Hu, Seema Sundara, and Jagannathan Srinivasan. 2009. Estimating aggregates in time-constrained approximate queries in Oracle. In *EDBT*. ACM.

[22] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2008. Scalable approximate query processing with the dbo engine. *TODS* 33 (2008).

[23] Srikanth Kandula et al. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*. ACM, 631–646.

[24] Albert Kim et al. Available at: data-people.cs.illinois.edu/needletail.pdf, 2016. Optimally Leveraging Density and Locality to Support LIMIT Queries. In *Technical Report*.

[25] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. SIGMOD.

[26] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering* 69, 1 (2010), 3–28.

[27] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation for Joins. In *SIGMOD*. ACM, 2121–2124.

[28] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.* 20, 12 (2014), 2122–2131. https://doi.org/10.1109/TVCG.2014.2346452

[29] S.L. Lohr. 2009. *Sampling: Design and Analysis*. Cengage Learning. https://books.google.com/books?id=aSXKXbyNlMQC

[30] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[31] Dominik Moritz, Danyel Fisher, Bolin Ding, and Chi Wang. 2017. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2904–2915.

[32] Frank Olken. 1993. *Random sampling from databases*. Ph.D. Dissertation. University of California at Berkeley.

[33] Kale Sarika Prakash and PM Joe Prathap. [n. d.]. Bitmap Indexing a Suitable Approach for Data Warehouse Design. *International Journal on Recent and Innovation Trends in Computing and Communication ISSN* ([n. d.]), 2321–8169.

[34] Rishi Rakesh Sinha and Marianne Winslett. 2007. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.* 32, 3 (2007), 16.

[35] Liwen Sun et al. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*. ACM, 1115–1126.

[36] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *VLDB* (2016).

[37] S.K. Thompson. 2012. *Sampling*. Wiley. https://books.google.com/books?id=-sFtXLIdDiIC

[38] John W Tukey. 1977. Exploratory data analysis. (1977).

[39] Wikipedia. 2016. Bitmap index. (2016). https://en.wikipedia.org/w/index.php?title=Bitmap_index&oldid=730569222 [Online; accessed 5-October-2016].

[40] Kesheng Wu et al. 2004. On the performance of bitmap indices for high cardinality attributes. In *VLDB*. VLDB Endowment, 24–35.

[41] K. Wu et al. 2009. FastBit: Interactively Searching Massive Data. *Journal of Physics Conference Series, Proceedings of SciDAC 2009* (June 2009).

[42] Kesheng Wu, Kamesh Madduri, and Shane Canon. 2010. Multi-level bitmap indexes for flash memory storage. In *IDEAS*. 114–116.