Progressive Indexing for Interactive Analytics

Michael Hohenstein
Technical University of Kaiserslautern
Erwin-Schrödinger-Straße, 67663 Kaiserslautern
Kaiserslautern, Germany
hohenstein@cs.uni-kl.de

ABSTRACT

Progressive Visual Analytics (PVA) is a recent paradigm in the realm of visualization. PVA is closely related to approximate query processing and streaming applications with a focus on real-time interactivity and features supporting a fluid and insightful experience for data analysts working on unknown data sources in an exploratory way. We want to observe the paradigm of progressive data science through the lens of database systems, more specifically considering existing technologies which can be exploited to support progressive systems. In this paper we briefly investigate the similarities of PVA to approximate query processing and present an indexing strategy which can be utilized in approximate and progressive environments without conflicting with its intrinsic requirements.

Keywords

progressive visual analytics, approximate query processing, indexing, sampling

1. INTRODUCTION

Progressive Visual Analytics (PVA) [4, 17] is a relatively new paradigm in the realm of visualization. It's main objective is to develop algorithms and infrastructure to support analysts in exploratory ad hoc data analysis. This means each query should return an (approximate) result within an upper time bound, so that data exploration can be considered a real-time process. Additionally, the analyst should be able to steer the query by tuning parameters of the computation. The key principle of the progressive paradigm is to instantly return an approximate result which is (progressively) updated in the background. Ideally some notion of (partial) re-use of earlier results that intersect with a live query should be in place to keep a certain *continuity* in a computation and further reduce the response time of later queries. Many recent papers on PVA ([3, 7, 8, 10]) have put the focus on features and requirements of progressive algorithms and have investigated potentials and future challenges of the paradigm.
Turkay et al. have discussed potentials and challenges of a progressive data science approach in the field of Machine Learning, Data Base Systems and Visualization [18]. They identify the following relevant areas of (recent) research in the DB community:

- Approximate Query Processing
- re-use of (partial) results and sampling that takes into account rare subpopulations
- self-adapting (data organization, indexing, etc.) data structures
- speculative query processing
- (progressive) data wrangling and cleaning

In this paper, we want to take a more detailed look at the application of progressive indexing [13] in the context of PVA. In particular, we will focus on a self-adapting index, touching the areas of approximate querying and sampling to support the requirements of progressive analysis tasks. The main contributions of the paper is to demonstrate the synergy between approximate (and by similarity of the concepts progressive) querying approaches and progressive indexing. Additionally, we present a sampling and indexing strategy useful in realizing systems for progressive query environments. The outline of our paper is as follows: First we want to give an overview about the paradigm of PVA and take a look at which existing data base technologies can be utilized to support the progressive paradigm. In sectiion 3 we present our approach of combining a strategy involving approximate query processing and construction of a progressive index to support aspects of "progressiveness"regarding fast, reliable intermediate results. We then present an evaluation of our efforts and in section 4. We close the paper with a conclusion and outlook on future work.

2. RELATED WORK AND BACKGROUND

The iterative nature of the progressive computation can be found in a number of already existing computational concepts. The most similar concepts that come to mind are *Streaming*, *Online*, and *Iterative* algorithms. In the realm of data base applications, the concept of *Online Aggregation* is closely related [9, 12]. Essentially, the field of PVA does not introduce a completely new computational paradigm, but describes a set of features a computational process should

32nd GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 01-03, 2021, Munich, Germany.

fulfill to to be classified as progressive, Stolper, Aupetit, and Fekete et al. [7] comprised a definition of features a progressive computation must possess. The computation should:

1. Be bounded on time and data.
2. Report intermediate outputs: a result, and measures of quality and progress.
3. Be bounded on latency.
4. Converge towards the *true* result.
5. Be controllable by a user during execution.

From the perspective of data base systems, this leads to the following ”progressive”requirements for a system or infrastructure designed for analytical components that adhere to the design goals mentioned above.

1. Suitable *sampling* algorithms with outlier detection.
2. Fast *approximate* algorithms.
3. Efficient (partial) *reuse* to bring continuity into a steered computation.
4. Metrics on *quality* and *progression* of the computation.
5. Guarantees on user given *time-bounds* to the first result and to new results.
6. Supporting structures to aid data analysts (Indexes, Meta Data, etc.).

In the context of this paper, we focus on requirements 1, 2, and 6: Sampling with outlier detection, approximate querying approaches, and indexing techniques to support progressive exploration tasks. More specifically, we propose a progressively built index which is constructed while a user poses fast approximate queries in a *real time* fashion. By real time we mean, that queries should return a result below the *interactivity threshold* of 500ms, as proposed by Liu et al [15]. For our approach, we investigated the work of Holanda et al. [13], which describes an index which is built progressively while querying a data base, and aim to integrate this indexing technique with Approximative Query Processing (AQP). Generally, there are two major categories of AQP approaches. online aggregation [12] and sampling [1]. Sampling approaches (e.g. BlinkDB [2], AQUA [16]) require preprocessing to deliver statistically significant results on lower populations, which clashes with the dynamic nature of exploratory data analysis. Online aggregation (e.g. CONTROL [11], DBO [14]) struggles to give reliable approximations for rare subpopulations of a large data set. To remedy these weaknesses, we studied the work of Galakatos et al. [9] who employs uniform sampling in tandem with rare subpopulation detection to amount for data skew and outliers. We use a similar approach, but combine it with parallel preprocessing to collect distributional metadata of the data set to be explored in order to improve later samples. We bridge the non-interactive preprocessing gap with uniform sampling (which requires no initial preprocessing), refined with rare subpopulation detection. As such, we are able to quickly answer queries with rare populations and data skew taken into account, while samples taken later accurately represent the real data.

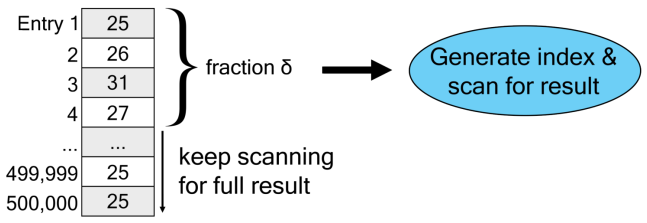
Entry 1
2
3
4
...
499,999
500,000
25
26
31
27
...
25
25
fraction δ
Generate index & scan for result
keep scanning for full result

Figure 1: Original Indexing Approach of Holanda et al. The entries are all tuples which will be returned for a running query. [13]

On a basic level, the indexer by Holanda et al. takes a fraction δ of the data of and adds them to a growing index (see Figure 1). The cost of index creation is ßmeared outöver the run time of queries issued to the data base. For a standard query environment, this means that initially queries will take rather long, as early inquiries to the data base can not benefit from indexes and have to scan all tuples sequentially.

More precisely, Holanda's approach is divided into three phases. In the *Creation Phase* a (standard, non approximate) query first performs an index lookup on the fraction of indexed data (ρ). Now, the not-yet-indexed $1 - \rho$ fraction of the original column is scanned while expanding the index by a chosen fraction δ of the column size. For the index creation Holanda et al. use one of four sorting algorithms: Quicksort, Bucketsort, Radixsort (MSD), and Radixsort (LSD). Depending on the utilized Indexing method, the partial index performs better for different querying scenarios. Once all data is indexed the approach enters the *Refinement Phase*. There, queries can be performed without scanning any non-indexed data. Additionally, the existing, rough index is refined, progressively converging towards a fully ordered index. Lastly, in the *Consolidation Phase* the ordered index is now progressively converted to a B+ tree for query efficiency. For an example, please refer to Figure 2 for a short overview of the quicksort indexing appproach of Holanda et al.

While useful in an exact, non approximate querying setting, Holanda states that progressive indexing techniques most likely will synergize well with approximate querying techniques, since the problem of initially longer running queries can be remedied by settling for faster, approximate results before an index structure exists.

3. PROGRESSIVE INDEXING IN THE CONTEXT OF APPROXIMATE QUERY PROCESSING.

The premise of progressive indexing is to iteratively create a database index on a previously unknown data source and ßmear out”the cost of doing so over several running queries. We want to explore if progressive indexing can be used alongside and/or benefit from a progressive querying approach. Since building a complete index can take a long time, doing so is kind of antithetical for an ad-hoc exploratory query session where fast (approximate) results on an unexplored and unprepared data source are the focus. The progressive indexing will incrementally build an index in parallel to interactive query sessions, speeding up certain queries, and also preparing the data base for queries requiring an

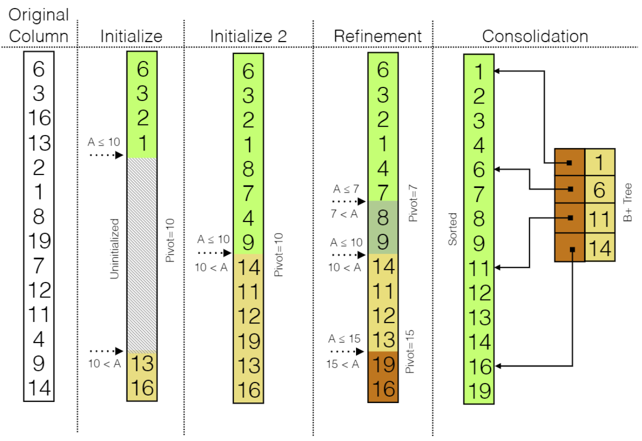

Figure 2: Example of progressive Indexing using Quicksort showing the Creation, Refinement, and Consolidation phases. [13]

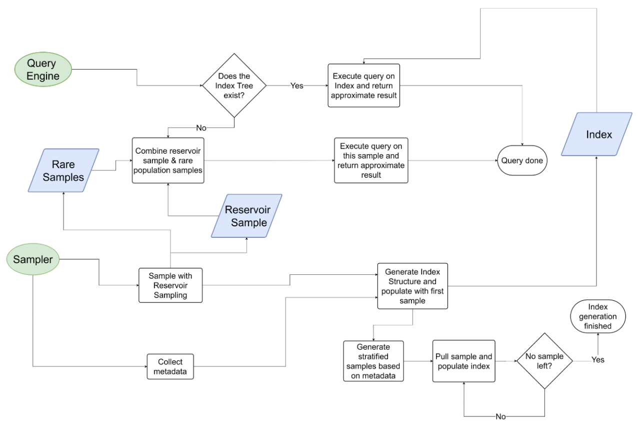
Query Engine
Does the Index Tree exist?
Yes
Execute query on Index and return approximate result
No
Combine reservoir sample & rare population samples
Execute query on this sample and return approximate result
Query done
Index
Rare Samples
Reservoir Sample
Sampler
Sample with Reservoir Sampling
Generate Index Structure and populate with first sample
Index generation finished
Collect metadata
Generate stratified samples based on metadata
Pull sample and populate index
No sample left?
Yes
No

Figure 3: Flowchart of the interaction between query engine and sampler.

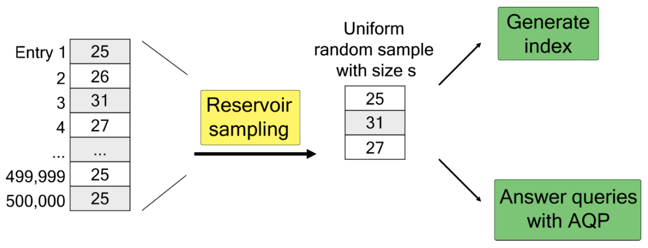

Figure 4: Modified Indexing Approach of Holanda et al. To account for Approximate Querying Techniques.

exact result. The synergy with approximate querying largely stems from the fact, that approximate / progressive queries don't have to scan the complete data base for a query which is fired against a data base with no index whatsoever. If only a subset of tuples is retrieved initially, the indexing can be done in a more dynamic manner alongside of a fluid "real timeüser interaction from the start. To explore this venue, we built a prototype comparing the suitability of various sampling / indexing techniques based on Holanda's work in the scope ([19]) of approximate querying applications. Roughly explained, our approach first enters a preparatory stage in which facilities for the progressive index are created and meta-data to ensure statistically significant samples is collected. After this initial stage the partial index is in place and populated in parallel to approximate querying sessions. To keep the system interactive, during the preparatory stage queries can be answered in parallel, albeit not with data skew or outliers taken into account.

The following features are our main contributions which differentiate our approach to the work of Holanda et al. First, we keep an absolute number s of tuples instead of a certain fraction δ of the complete data set to determine how many tuples to index, since we have no clue how many rows the data set to be inspected has. Also, not only do we partially index the complete set per query, we also employ approximate query processing and trade accuracy of results for speed concerning the answering of queries during all stages of querying. In the same vein, we supplement our sampling approaches with rare population indexing to improve the statistical significance of samples taken from the original data.

For a high-level overview of the architecture of the query and sampling engine, please refer to Figure 3.

3.1 Initial Sampling and Preprocessing

Since in an approximate environment the cardinality of the complete data base is unknown, it is impossible to just determine some fraction δ of the complete data set which should be indexed per query. Since for our case we always take (increasing) samples of the complete data, we index s many tuples, where s is the size of the sample (See Figure 4). A nice side effect of this approach is, that an initial partial index is available much earlier, as the query only processes a subset of the complete data set and does not have to scan the complete data set.

Approximate results require a random sample to be statistically relevant. Taking just the first δ rows as a sample like in the original implementation would not return good results. As Galakatos et al. point out [6], we require some initial knowledge about a data set to be able to reach some modicum of statistical significance with standard sampling methods (like the total row count for example). To solve this dilemma, we utilize Reservoir Sampling [20] to create the first sample. Since reservoir sampling misses any skew or outliers in the data, we combine the Reservoir Sampling technique with preprocessing of the data to be streamed to make the *following* samples more representative of the real data set. We count the number of rows of the data set as and record the count of attribute groups to get meta data about a columns distribution and the replacement probability for each row, as shown in Algorithm 1. While a preprocessing step might seem to go against to the requirements of the progressive paradigm, we deem it acceptable as queries can already be answered during the preprocessing, albeit with less accurate results.

3.1.1 Rare Population Indexing

Together with the reservoir sample, we build another sample that takes rare attribute values into account. To do this, we designate a certain fraction θ as a threshold for a value to count as rare. For this we use the current row count and attribute distribution gained during the reservoir preprocessing. Any attribute appearing for a fraction of less than θ of the row count will be sampled separately. Since the row count increases during iteration, we have to continuously check if we still consider an attribute rare. Once an attribute is deemed rare, it will be removed from the rare sample pool if it appears more often than $(\theta * 2) * current\ rowcount$. Using this method will provide sufficient entries in the sample for rare groups and thus more interesting results, even in the early phases of sampling. The rare samples however will be discarded when the system switches to static sampling to guarantee the correct distribution and proper randomness (See Algorithm 1).

Algorithm 1: Modified Reservoir Sampling

```
Input: Data stream to record distribution from.
Output: Random uniform sample of dataset, random
        samples from rare populations.
rowcount ← 0;
rareThreshold ← 0;
groupCounts ← initialize dictionary;
sample ← initialize sample;
for each row in datastream do
    groupCounts ← occurenec of att. value;
    rowCount += 1;
    if length of sample < max. sample size then
        add row to sample;
    else
        replacement prob. ← sample size / rowcount;
        if value gets replaced then
            replacedRow ← rand. row in curr. sample;
            replace row with new row;
            if replacedRow is rare then
                resample replaced row in separate
                  sample
            end
        else
            if row is a rare entry then
                resample row in separate sample
            end
        end
    end
    rareThreshold ← rowCount * $\theta$;
end
```

3.2 Early Query Executions

Early queries encompass all queries that are executed before any index is built. The set of tuples these queries are executed on is a combination of the existing random uniform reservoir sample and parts of the random samples for rare populations. If the tuples of the reservoir sample contain too few of any rare attribute value, tuples contained in the separate index of rare populations will be randomly added so that they are not overlooked. This will produce more reliable results than straight reservoir sampling, even if they are somewhat biased. Mind this mode of querying is only temporary, as it is only included to bridge the time needed to preprocess the data set and create the initial index structure which incorporates a stratified sampling approach to take into account data skew and rare populations.

3.3 Index Generation

After having created an initial random sample and a full pass over the data to collect distributional metadata, a par-

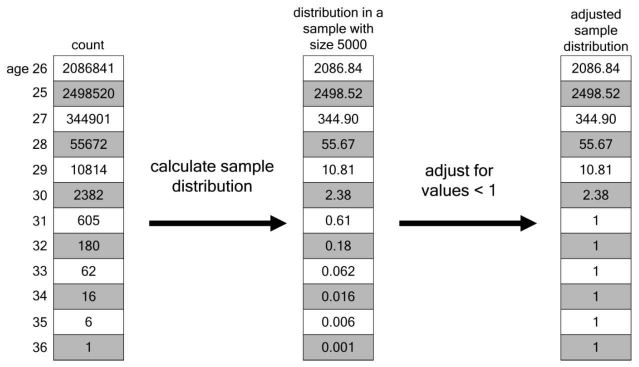

Figure 5: Creation of the Adjusted Stratified Sample based on the distribution of the age column

tial index tree is created. For this we have implemented the Progressive Radixsort (MSD) method, similar to Holanda et al. [13]. From the initial preprocessing, we create a B+ tree index structure for each column in accordance of the amount of different attribute instances we observed. Into this index tree the original reservoir sample is then inserted. Now, this index tree will bee populated with more samples over time, utilizing any down-time during a user session. As mentioned earlier in this section, we collected metadata during the initial sampling step about the distribution of the data. Thus, all future samples which are used to populate the index structure can be chosen so that they reflect the distribution of the original data using a stratified sampling [5] approach. Data samples are thus pulled randomly from their respective groups and not just randomly overall. Once an (approximate) query is issued, the progressive indexing is paused, and the query is only answered by tuples already indexed. If an extreme skew is present, it can easily happen that samples would include less than one entry to reflect the correct distribution. We opt to include at least one of these very rare datapoints with each sample as illustrated in Figure 5, since outliers often are a point of interest for exploratory data analysis. The distribution in the index is thus slightly off in the beginning, but since the implementation keeps pulling samples continuously, this error will correct itself quickly over time. This also guarantees that enough rare entries are available early to create meaningful approximations. When the samples are generated, they are pulled and indexed sequentially until the data is exhausted. At this point, the index is fully built and all future queries will run with the full index. Note, that while later approximate queries take *longer* due to the ever increasing number of stratified tuples entered into the index, the gain in result accuracy is usually worth it. While Holanda's original approach only refines the coarse index into a B+ tree after *all* data has been indexed, we chose to refine the index immediately after indexing a new fragment of the original data. The small overhead of immediate refinement to a full B+ tree allows for much faster queries than on a coarse index. There queries would quickly reach undesirable delays if issued over large indexes. If the index becomes too large that querying it completely becomes too costly, we can resort either to querying only a statistically relevant sample of the *index* or to pruning the index to a subset of the original data. To keep these approximate queries or pruned indexes statisti-

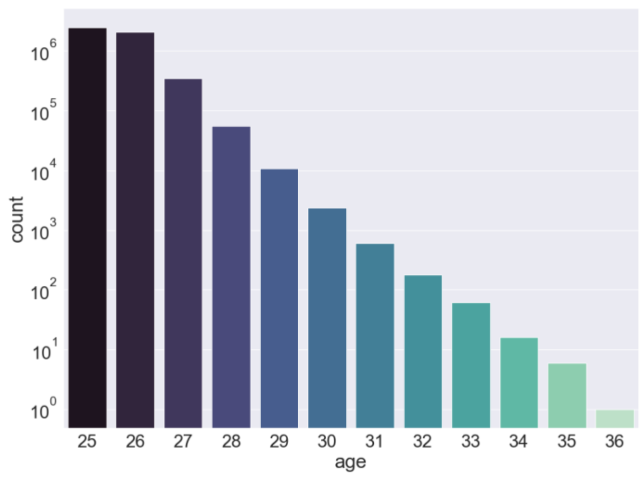

Figure 6: Distribution of the age value of the test data set.

cally relevant, we reference the distributional metadata to create an appropriate approximation of the original data. It should be obvious, that at this point, the index structure is not only useful for approximate queries, but will also serve in scenarios where exact results are required.

4. EVALUATION

Since it is not really interesting if the queries themselves are faster for an approximate setting instead of a contemporary setting, we will focus on comparing the time until the progressive indexing converges to a full query compared to the approach of Holanda et al., and how well the stratified index manages to account for data skew while uncompleted.

4.1 Setup

For the evaluation, we have implemented the system in Python version 3.8.3. The experiments were run on a machine with an Intel 6700k processor at 4.4GHz and 32GB of DDR4 RAM at 2133MHz.

4.2 Data

We have generated a dataset with five million rows containing data about the age, salary and received tips of waiters. To test the indexing prototype, we made sure to integrate different value distributions and amount of skew into the data. The tip tuples are uniformly distributed. The distribution of the age and salary values (see Figures 6 and 7) is each skewed towards a particular value range.

4.3 Time until Index Convergence

For this experiment, we implemented progressive indexing in a similar fashion as Holanda et al. demonstrated in their approach (specifically the Radix MSD variation in its base form) and ran it on the same data set. Only one index on the ‘age’ column was constructed. Each second we repeatedly issued a simple query (age $\geq$ 32) until the index converged. We did not implement the adaptive indexing budget proposed by Holanda et al., but instead set the to-be-indexed fraction with each query to 10%. The sample size for our version was again set to 5000. In Figure 8 we can see that a progressive indexing approach greatly benefits from approximate querying techniques. The base version takes 404

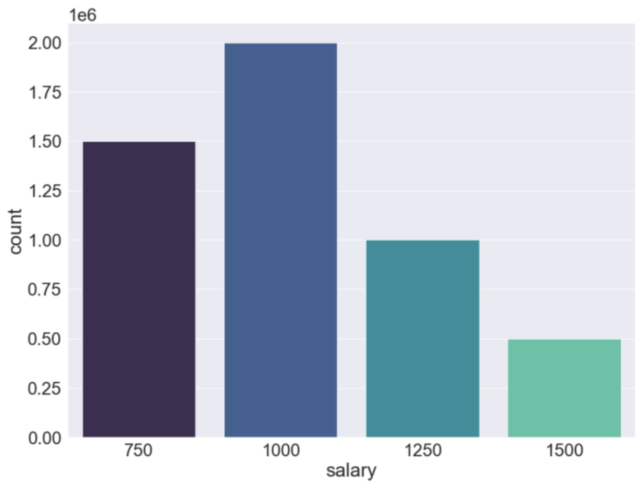

Figure 7: Distribution of the salary value of the test data set.

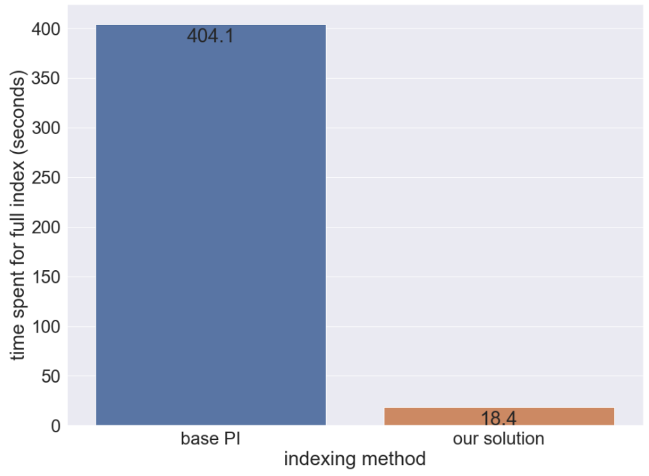

Figure 8: Duration for the B+ index reaching full convergence while repeatedly issuing the simple query (age $\geq$ 32).

seconds to a fully converted index, while our approach only takes 18 seconds. We thus reach index convergence roughly 22 times faster than the original approach. This is mostly due to the fact that the original approach relies on queries always ranging over the complete data base to further generate the index. As a result, the index cannot be constructed further until the current query has finished processing all tuples. Our solution on the other hand utilizes the much shorter time to return of approximate queries to be able to quickly construct an index.

4.4 Accounting for Data Skew

In this experiment, we looked at how well our sampling approach can handle data skew. For this, we ran a `count('age')` query run once with an initial reservoir sample generated in the first step before a stratified index is built, and afterwards on a reservoir sample including rare subpopulation sampling. The sample size for this test was 5000. Figure 9 shows the pitfall of standard reservoir sampling. Since the values 25 and 26 represent the bulk of the entries, other entries are easily missed. Especially the rare outliers are excee-

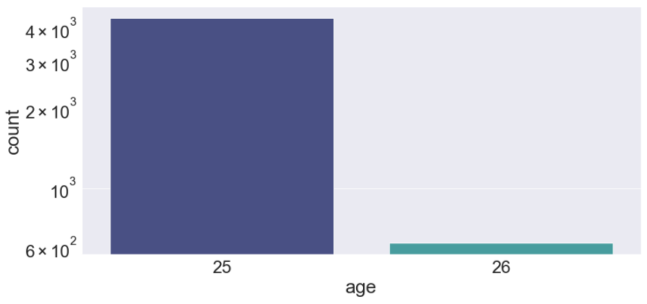

Figure 9: Distribution of a sample of salary values taken with reservoir sampling.

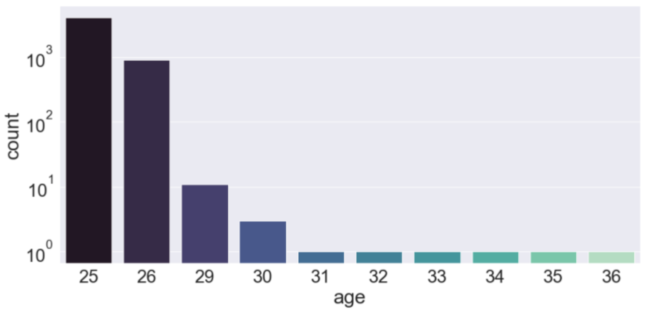

Figure 10: Distribution of a sample of salary values taken with stratified sampling.

dingly unlikely to be included in the initial sample. Figure 10 shows the impact rare population sampling strategy can have on a sample of the same size. Even exceedingly rare outliers (age 30-36) are included in the sample. While not correct, this sample composition is a much better representation of the real data set. Here we can see the advantage of including rare population sampling in addition to reservoir sampling until we have a true stratified sample. When the stratified index structure is built later, data skew largely stops being a problem as we now know the distribution of values. As shown in Figure 5, the distribution is not exact simply because we can not include fractions of tuples, but good enough to give a quite accurate picture of the complete data.

5. CONCLUSION AND OUTLOOK

For this work we prototyped a progressive indexing approach usable in an approximate querying environment inspired by a Progressive Indexing approach by Holanda et al. [13]. The AQP environment leads to significantly lower query execution times, even when the data is not fully indexed. To make these approximate results more significant, we combined reservoir sampling, meta data collection (rare population sampling) and a stratified sampling approach to remedy for the intrinsic inaccuracy of approximate queries. Our evaluation results confirm Holanda's assumption that the progressive indexing approach benefits greatly from the much faster time to return of approximate queries on non-indexed data. For future work on this topic, we identified two major areas of potential improvements: Including an indexing budget, and expanding the stratified sampling strategy. In the original approach by Holanda et al., an indexing budget was used to automatically adjust how much of the query time should be used for indexing. We could imagine a similar solution for automatically determining the sample size for the progressive index in an approximate query environment. For the stratified sampling, we chose a very simple solution to determine how the samples should be generated by just relying on the skew of one column. In a real environment, this might not be viable since other columns might be just as interesting. Creating a more refined scheme to account for skew found in several attributes would further increase the reliability of the partial results returned by approximate queries with our proposed indexing strategy.

6. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 487–498. ACM, 2000.

[2] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. *CoRR*, abs/1203.5485, 2012.

[3] M. Angelini, T. May, G. Santucci, and H.-J. Schulz. On quality indicators for progressive visual analytics. 06 2019.

[4] M. Angelini, G. Santucci, H. Schumann, and H. Schulz. A review and characterization of progressive visual analytics. *Informatics*, 5(3):31, 2018.

[5] M. P. Cohen. Stratified sampling. In M. Lovric, editor, *International Encyclopedia of Statistical Science*, pages 1547–1550. Springer, 2011.

[6] K. Dursun, C. Binnig, U. Çetintemel, and T. Kraska. Revisiting reuse in main memory database systems. *CoRR*, abs/1608.05678, 2016.

[7] J. Fekete, D. Fisher, A. Nandi, and M. Sedlmair. Progressive data analysis and visualization (dagstuhl seminar 18411). *Dagstuhl Reports*, 8(10):1–40, 2018.

[8] J. Fekete and R. Primet. Progressive analytics: A computation paradigm for exploratory data analysis. *CoRR*, abs/1607.05162, 2016.

[9] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[10] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos. Progressive similarity search on time series data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019*, 2019.

[11] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *Computer*, 32(8):51–59, 1999.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182. ACM Press, 1997.

[13] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt. Progressive indexes: Indexing for

interactive data analysis. *PVLDB*, 12(13):2366–2378, 2019.

[14] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 725–736. ACM, 2007.

[15] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[16] T. K. Sellis. Review - the aqua approximate query answering system. *ACM SIGMOD Digit. Rev.*, 2, 2000.

[17] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, 2014.

[18] C. Turkay, N. Pezzotti, C. Binnig, H. Strobelt, B. Hammer, D. A. Keim, J. Fekete, T. Palpanas, Y. Wang, and F. Rusu. Progressive data science: Potential and challenges. *CoRR*, abs/1812.08032, 2018.

[19] M. van den Berg. Applying progressive indexing in the context of approximate query processing. 2021.

[20] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.