Revisiting Reuse for Approximate Query Processing

Alex Galakatos Andrew Crotty Emanuel Zgraggen Carsten Binnig Tim Kraska*
Department of Computer Science, Brown University
{firstname_lastname}@brown.edu

ABSTRACT

Visual data exploration tools allow users to quickly gather insights from new datasets. As dataset sizes continue to increase, though, new techniques will be necessary to maintain the interactivity guarantees that these tools require. Approximate query processing (AQP) attempts to tackle this problem and allows systems to return query results at "human speed." However, existing AQP techniques start to break down when confronted with ad hoc queries that target the tails of the distribution.

We therefore present an AQP formulation that can provide low-error approximate results at interactive speeds, even for queries over rare subpopulations. In particular, our formulation treats query results as random variables in order to leverage the ample opportunities for result reuse inherent in interactive data exploration. As part of our approach, we apply a variety of optimization techniques that are based on probability theory, including new query rewrite rules and index structures. We implemented these techniques in a prototype system and show that they can achieve interactivity where alternative approaches cannot.

1. INTRODUCTION

The widespread popularity of visual tools [31, 8, 36, 6] for interactive data exploration has empowered domain experts in a broad range of fields to make data-driven decisions. However, a key requirement of these tools is the ability to provide query results at "human speed," even over large datasets. For example, a recent study [24] demonstrated that delays longer than even $500ms$ can negatively impact user activity, dataset coverage, and insight discovery.

As dataset sizes continue to increase, traditional DBMSs designed around a blocking query execution paradigm cannot scale to meet these interactivity requirements. Techniques that pre-aggregate data on top of a DBMS (e.g., materialized views, data cubes [10]) can significantly reduce query latency, but they require extensive preprocessing and suffer from the curse of dimensionality. Unfortunately, attempts to overcome these problems (e.g., imMens [25], NanoCubes [23]) usually restrict the number of attributes that can be filtered at the same time, severely limiting the possible exploration paths.

Rather, in order to achieve low latency for ad hoc queries, new systems targeting interactive data exploration must instead rely upon *approximate query processing* (AQP) techniques, which provide query result estimates with bounded errors. Many AQP systems use some form of biased sampling [1] (e.g., AQUA [2], BlinkDB [3], DICE [19]), but these approaches usually require a priori knowledge of the workload or substantial preprocessing time. While useful for many applications, this approach is at odds with interactive data exploration, which is characterized by the free-form analysis of new datasets where queries are typically unknown beforehand.

On the other hand, AQP systems that perform online aggregation [13] (e.g., CONTROL [12], DBO [18], HOP [5], FluoDB [35]) can generally only provide good approximations for queries over the mass of the distribution, while queries over rare subpopulations yield results with loose error bounds or runtimes that vastly exceed the interactivity threshold. Since the exploration of rare subpopulations (e.g., high-value customers, anomalous sensor readings) often leads to the most significant insights, online aggregation falls short when confronted with these types of workloads.

We therefore argue that neither biased sampling nor online aggregation is ideal in this setting, since systems for interactive data exploration must provide high quality approximate results for any query—even over rare subpopulations—at interactive speeds without requiring foreknowledge of the workload or extensive preprocessing time. While seemingly impossible, these systems have the opportunity to leverage the unique properties of interactive data exploration. Specifically, visual tools have encouraged a more conversational interaction paradigm [15], whereby users incrementally compose and iteratively refine queries throughout the data exploration process. Moreover, this style of interaction also results in several seconds (or even minutes) of user "think time" where the system is completely idle. Thus, these two key features provide an AQP system with ample opportunities to (1) reuse previously computed (approximate) results across queries during a session and (2) take actions to prepare for possible future queries.

To take advantage of these opportunities, we present an AQP formulation that maximizes the reuse potential of approximate results across queries during an exploration ses-

*Work done at Brown, currently at Google: kraska@google.com

Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.

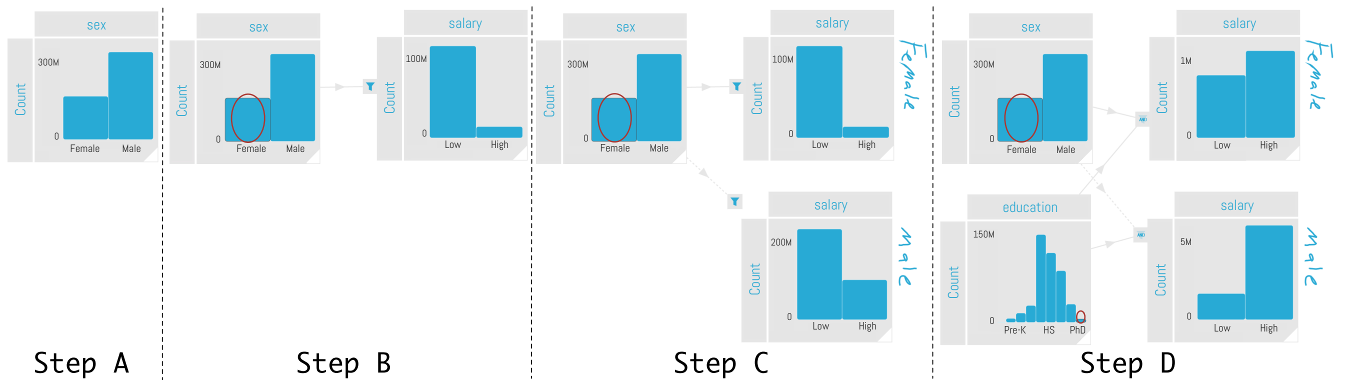


Figure 1: An example exploration session where the user is trying to understand which attributes affect an individual's *salary*. First, the user examines the *sex* attribute (Step A) to view the distribution of 'Male' and 'Female' values. The user then links *sex* to *salary* and filters the *salary* distribution to only 'Female' (Step B). A second *salary* bar chart with a negated link (dotted line) compares the salaries for 'Male' and 'Female' subpopulations (Step C). Finally, linking *education* to each of the *salary* visualizations and selecting only individuals with a 'PhD' shows the respective salaries of highly educated males and females (Step D).

sion while at the same time permitting formal reasoning about error propagation. Our approach is a reformulation of online aggregation that treats aggregate query results as random variables, unlocking a completely new set of query rewrite rules based on probability theory. To support queries over rare subpopulations, we additionally present the *Tail Index*, a low-overhead partial index that integrates with our AQP formulation and enables low-error result approximation at interactive speeds.

In summary, we make the following contributions:

- We propose an AQP formulation that treats aggregate query answers as random variables to enable reuse of approximate results with formal reasoning about error propagation across overlapping queries.
- To enable exploration of rare subpopulations, we present (1) novel query rewrite rules based on probability theory and (2) Tail Indexes that help to achieve low-error approximate results.
- We implemented our techniques in a prototype Interactive Data Exploration Accelerator [7] (IDEA), and our benchmarks use real world datasets to demonstrate that we can achieve interactive latencies in many cases where alternative approaches cannot.

2. OVERVIEW

In interactive data exploration tools, many types of visualizations (e.g., bar charts, histograms) serve to convey high level statistics about datasets, such as value distributions or relationships between attributes. As previously mentioned, interaction with these tools is conversational in nature, as users incrementally build up and iteratively refine queries by manipulating visualizations. This exploratory process precludes techniques that require a priori knowledge of the workload, since users compose unforeseen queries in order to derive new insights from the data.

As an illustrative example of this process, consider the sample exploration session shown in Figure 1, which is drawn from a past user study [7]. In the exploration session, the user is analyzing data from the 1994 US census [22] using Vizdom [6] in order to understand which attributes affect an individual's annual salary. While online aggregation can be used to provide progressive results for these visualizations, it treats each query as "one-shot" (i.e., a slightly modified version of a past query is treated as a completely new query) and therefore does not effectively model the conversational interaction paradigm. At the same time, existing techniques for query result reuse [33, 16, 26, 9] do not consider partially computed results.

In this section, we first introduce our AQP formulation (Section 2.1), which is an extension of online aggregation that treats aggregate query results as random variables. We then describe how our prototype IDEA implementation executes an aggregate query using this formulation (Section 2.2), as well as how we quantify the error associated with approximate results returned to the user (Section 2.3).

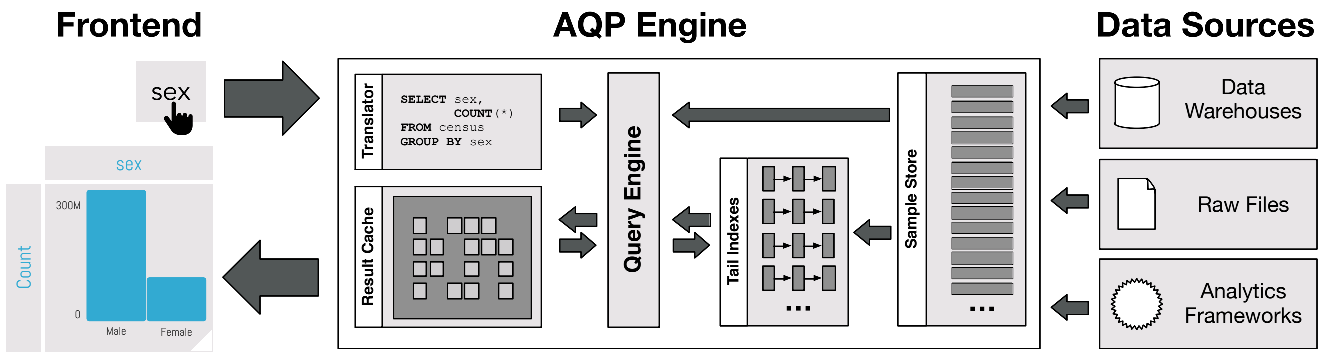
2.1 Formulating a Simple Query

The simplest type of visualization we consider is a bar chart that expresses the count of each attribute value. For example, Step A of Figure 1 shows a bar chart of the *sex* attribute, which can be represented by the following SQL query:

```
SELECT sex, COUNT(*)
FROM census
GROUP BY sex
```

One way to model a group-by attribute is to treat it as a categorical random variable X, where X can assume any one of the possible outcomes from the sample space Ω_X. For example, a random variable modeling the *sex* attribute can take on an outcome in the sample space $\Omega_{sex} = \{Male, Female\}$. The discrete probability distribution θ_X represents the likelihood that X will assume each outcome, with θ_x denoting the probability that X will assume outcome x. In the example query, for instance, the probability of observing a 'Male' tuple is $\theta_{Male} \approx 2/3$.

As previously mentioned, an AQP engine needs to use techniques to compute an approximation $\hat{\theta}_X$ of the true θ_X, since processing a large dataset in a blocking fashion can easily exceed interactivity requirements. Therefore, for each distinct value x in Ω_X, *maximum likelihood estimation* (MLE) can produce an approximation $\hat{\theta}_x^{MLE}$ of the true



```
SELECT sex,
       COUNT(*)
FROM census
GROUP BY sex
```

Figure 2: Full execution path through the system to produce the *sex* bar chart from Step A of Figure 1.

θ_x by scanning a random sample of the underlying data to compute:

$$\hat{\theta}_x^{MLE} = \frac{n_x}{n} \tag{1}$$

Equation 1 shows that each θ_x is approximated by dividing n_x (i.e., the number of observed instances of x in the sample) by the total sample size n. Intuitively, this approximation represents the estimated *frequency* of x in the dataset. Multiplying a frequency estimate $\hat{\theta}_x$ by the total number of tuples N in the full dataset will therefore yield an approximate count for x:

$$count(x) \approx N\hat{\theta}_x \tag{2}$$

These techniques can be extended to support histograms (i.e., a count grouped by a continuous attribute like *age*) by partitioning the domain of the group-by attribute into finite bins. Then, these bins can be treated the same as a nominal attribute. We use similar techniques to compute other aggregates (e.g., average, sum) and more complex visualizations (e.g., heatmaps), as well as queries involving joins, all of which are discussed in Section 5.

2.2 Executing a Query

One of the key contributions of this paper is to show how query processing changes in the context of our AQP formulation. To understand how to execute a query and produce a visualization using the formulation presented in the previous section, consider again the example query representing the *sex* bar chart (Step A in Figure 1). Figure 2 shows the full execution path through the system, from issuing the query in the frontend to the final progressive result visualization.

As the first step in an exploration session, a user will typically connect to a completely new data source (e.g., the census data stored in a data warehouse). Upon connecting, the AQP engine immediately begins to populate the *Sample Store* by streaming in tuples from the underlying data source. Since our techniques rely on a random sample in order to get a good approximation, we use methods to access the underlying data source in a random order (e.g., reservoir sampling, randomized index traversal [27]). The Sample Store caches a subset of tuples from the data source (or the entire dataset if memory permits) that serves as the basis for answering queries using our AQP techniques.

When the user creates the visualization shown in Step A of Figure 1 in the frontend by dragging out the *sex* attribute to the screen, the *Translator* receives the visualization parameters (i.e., attribute, aggregate function, selection predicates) and generates a corresponding SQL query. The *Query Engine* then executes this query by spawning several executor threads, which draw tuples from the Sample Store to compute an approximate result.

When reading these tuples, the Query Engine uses our AQP formulation to compute $\hat{\theta}_{sex}$ (using Equation 1) for all attribute values in *sex* (i.e., $\hat{\theta}_{Male}$ and $\hat{\theta}_{Female}$). The resulting probabilities (e.g., $\hat{\theta}_{Male} \approx 2/3$) are stored in the *Result Cache*. The Query Engine can leverage these cached results (e.g., Step B can reuse $\hat{\theta}_{Female}$) for computing the results to future queries, explained further in Section 3.

In addition to computing the frequencies, the Query Engine also constructs *Tail Indexes*, low-overhead index structures built on a subset of tuples in the Sample Store that help provide high-quality approximations for subsequent queries on rare subpopulations. We discuss how and when to build indexes in Section 4.

Finally, the frontend polls the Result Cache to receive the current approximation for each of the group-by values in the specified visualization.

2.3 Quantifying Result Error

As previously mentioned, the Query Engine updates the approximate results that are stored in the Result Cache as more data is processed. Therefore, we need a way of quantifying the uncertainty, or *error*, associated with each approximate result.

In the context of our AQP formulation, the previously described estimators are said to be asymptotically normal with the sample size n; that is, the estimator $\hat{\theta}_x$ is equal to the actual θ_x plus some normally distributed random noise given a sufficiently large sample. The *standard error* approximates this deviation of the observed sample from the true data. Formally, the normalized standard error of $\hat{\theta}_x$ is given by the square root of the variance of a categorical random variable divided by the sample size n, then normalized by $\hat{\theta}_x$:

$$error(x) = \frac{1}{\hat{\theta}_x}\sqrt{\frac{\hat{\theta}_x(1-\hat{\theta}_x)}{n}} \tag{3}$$

Note that Equation 3 does not consider the total data size N, which implies that, for smaller N, we might overestimate the error. We can address this issue using common techniques such as the finite population correction factor, but we leave this extension to our formulation for future work.

In order to calculate the error for a query grouped by an attribute X (e.g., the query to compute the *sex* bar chart in Step A of Figure 1), we compute the sum of the relative standard errors (Equation 3) for all attribute values $x \in \Omega_X$. Note, however, that metrics other than the sum (e.g., average, max) or alternative definitions [20] can be used to quantify result error.

3. APPROXIMATE RESULT REUSE

As previously mentioned, our AQP formulation is designed to leverage the unique reuse opportunities that exist in interactive data exploration sessions. In this section, we first show how our formulation extends to queries with selections and allows us to integrate the results of past queries that have already been (partially) computed. To facilitate result reuse, we also introduce several new rewrite rules based in probability theory that enable the Query Engine to optimize queries similarly to how a traditional DBMS uses relational algebra rules to optimize SQL queries. Finally, since we are reusing approximate results, we discuss the intricacies of error propagation and how to choose among several possible rewrites.

3.1 Queries with Selections

Section 2 showed how we can use our AQP formulation to express a single bar chart. One of the key features of data exploration tools, though, is the ability to create complex *filter chains* in order to explore specific subpopulations; that is, visualizations act as active controls and can be linked together, where selections in upstream visualizations serve as filters for downstream visualizations. For example, Step B of Figure 1 shows a filtered *salary* bar chart for only the 'Female' subpopulation, which translates to the following SQL query:

```
SELECT salary, COUNT(*)
FROM census
WHERE sex = 'Female'
GROUP BY salary
```

To answer this query, we need to estimate the probability of each possible *salary* value for only the 'Female' subpopulation. More formally, given a group-by attribute X and a selection attribute $Y = y$, we are trying to estimate the *joint probability* $\theta_{x,y}$ for every x value in Ω_X. By extending Equation 1 from Section 2.1, we can approximate each $\theta_{x,y}$ using the MLE:

$$\hat{\theta}^{MLE}_{x,y} = \frac{n_{x,y}}{n} \tag{4}$$

3.1.1 Reformulating Joint Probabilities

Although formally correct, the estimator $\hat{\theta}^{MLE}_{x,y}$ given in Equation 4 introduces a new issue that does not arise in the simple case of estimating θ_x. Namely, computing $\hat{\theta}^{MLE}_{x,y}$ the same way as $\hat{\theta}^{MLE}_{x}$ (described in Section 2.2) considers all estimates as independent, which could lead to inconsistencies [34] across different visualizations. For instance, an inconsistency could arise in Step B of the example exploration session if the sum of the two estimated *salary* values exceeded the total number of all 'Female' tuples, which is clearly impossible.

Since our AQP formulation treats query results as random variables, we can avoid this issue by leveraging the *Chain Rule* from probability theory to rewrite $\hat{\theta}_{x,y}$ into a different form:

$$\hat{\theta}_{x,y} = \hat{\theta}_{x|y}\hat{\theta}_{y} \tag{5}$$

In this new form, we can therefore estimate $\theta_{x,y}$ by reusing the previously computed estimate $\hat{\theta}_y$ and computing a new estimate for $\theta_{x|y}$, again using the MLE:

$$\hat{\theta}^{MLE}_{x|y} = \frac{n_{x|y}}{n_y} \tag{6}$$

For example, we can apply the Chain Rule to rewrite the query from Step B of Figure 1 to be conditioned on $sex = \text{'}Female\text{'}$, where we can estimate the joint probability by computing $\hat{\theta}^{MLE}_{salary|Female}$ and reusing the previously computed $\hat{\theta}_{Female}$. Note that we can equivalently rewrite the joint probability as $\hat{\theta}_{x,y} = \hat{\theta}_{y|x}\hat{\theta}_x$, and the Query Engine is free to choose whichever alternative produces the lowest error result in the shortest time (discussed further in Section 3.3). Again, since we multiply the conditional probability by the selected subpopulation (e.g., 'Female'), our estimates are no longer independent; that is, the sum of the estimates for values within a subpopulation must be strictly smaller than the estimate of the subpopulation itself, thereby addressing the visual inconsistency issue.

3.1.2 Very Rare Subpopulations

As previously mentioned, queries over rare subpopulations occur frequently in interactive data exploration settings, since they often produce the most interesting and valuable insights. Although the rewritten MLE that uses the conditional probability will provide a good approximation for the mass of the distribution, it is limited by the number of relevant tuples that are observed, yielding results with high error in the tails of the distribution.

To solve this problem, we can reuse the prior information available in the Result Cache in order to get a better estimate faster than scanning the Sample Store and filtering only for the rare subpopulation. Since the MLE cannot incorporate these priors, we can extend our formulation from Equation 1 to be able to include any available prior information. A *maximum a posteriori* (MAP) estimator incorporates an α term to represent prior information, where the estimate represents the mode of the posterior Dirichlet distribution:

$$\hat{\theta}^{MAP}_{x,y} = \frac{n_{x,y} + \alpha_{x,y} - 1}{n + \alpha - 2} \tag{7}$$

If no prior information is used (i.e., all possible parameters are equally likely), or if n approaches infinity (i.e., the impact of the prior goes to zero), then the MAP estimate is equivalent to using the MLE. Each of these scenarios occur, respectively, when we have no prior information about a distribution (e.g., as in Step A of the example before the user has issued any queries), or when a selection occurs over the mass of the distribution, thereby guaranteeing a large n in a short amount of time. Therefore, the MAP estimate can handle cases with very rare subpopulations by incorporating a prior while also providing the same performance as an MLE over the mass of the distribution.

For example, in the previous query (Step B), we are trying to compute $\hat{\theta}_{salary|Female}$. Since the Result Cache already contains an estimate for the different *salary* subpopulations (i.e., $\hat{\theta}_{High}$ and $\hat{\theta}_{Low}$), we can leverage these already computed estimates by providing them as a prior to their respective conditional estimates.

Unfortunately, when trying to incorporate a prior, the weighting is the hardest part [30], since overweighting will ignore new data while underweighting might produce a skewed estimate based on too few samples. Therefore, we pick a small α proportional to the expected frequency of the estimate, such that the significance of the prior will quickly diminish as more data is scanned unless the subpopulation is very rare, in which case the prior will help to smooth an estimate based on a tiny sample. As in Equation 3, we can use the finite population correction factor to reduce the impact of the prior as more data is scanned (e.g., a prior should have no impact after scanning all tuples). Again, we leave this optimization for future work.

3.2 Query Rewrite Rules

The previous section showed how to apply the Chain Rule in order to rewrite a query represented as a joint probability into a form that can provide a better estimate by reusing previously computed results. Thus, a natural follow up question is: can we leverage other rules from probability theory to rewrite different types of queries in order to maximize reuse potential?

Similar to a DBMS query optimizer that uses relational algebra rules to rewrite queries, our AQP formulation unlocks a whole new set of rewrite rules based on probability theory. By using these rewrite rules, the Query Engine can often leverage past results to answer some queries much faster than having to scan the Sample Store in order to compute a result from scratch. In fact, as our experiments show (Section 6), rewrite rules can even return results for certain queries almost instantaneously.

In the following, we describe example queries along with corresponding opportunities to apply rewrite rules during the query optimization process. Note that these rewrite rules are only possible due to our AQP formulation, and techniques that treat queries as "one-shot" (e.g., online aggregation) would need to resort to scanning the Sample Store in order to compute the result for each new query.

3.2.1 *Bayes' Theorem*

In many cases, users often wish to view the distribution of an attribute X filtered by some other attribute Y. Then, in order to get a more complete picture, they will reverse the query to view the distribution of Y for different subpopulations of X. Based on this exploration path (i.e., X filtered on Y switched to Y filtered on X), the Query Engine can use *Bayes' Theorem* to compute an estimate without having to scan any tuples. Formally, Bayes' Theorem relates the probability of an event based on prior knowledge of related conditions:

$$\hat{\theta}_{y|x} = \frac{\hat{\theta}_{x|y}\hat{\theta}_y}{\hat{\theta}_x} \tag{8}$$

For example, suppose that the user reverses the direction of the link in Step B (i.e., *sex* is now filtered by *salary*) and selects the 'High' bar, yielding the following SQL query:

```sql
SELECT sex, COUNT(*)
FROM census
WHERE salary = 'High'
GROUP BY sex
```

In this example, the Result Cache has (partially) complete estimates for all attribute values for the *salary* attribute conditioned on the attribute values for *sex* (i.e., $\hat{\theta}_{Low|Female}$, $\hat{\theta}_{High|Female}$, $\hat{\theta}_{Low|Male}$, $\hat{\theta}_{High|Male}$) from the query in Step B. Using Bayes' Theorem therefore allows us to leverage the previous results for (1) $\hat{\theta}_{sex}$, (2) $\hat{\theta}_{salary}$, and (3) $\hat{\theta}_{salary|sex}$ to instantly compute $\hat{\theta}_{sex|salary}$ without needing to scan any tuples from the Sample Store.

3.2.2 *Law of Total Probability*

Another common exploration pattern is to view the distribution of an attribute X for different subpopulations of Y, switching between filtering conditions to observe how the distribution of the downstream attribute changes. Recall that categorical random variables have the constraint that all outcomes are mutually exclusive (i.e., the probabilities sum to one). We can therefore leverage the *Law of Total Probability* to rewrite queries involving different selections over mutually exclusive subpopulations in order to reuse past results. The Law of Total Probability defines the probability of an event x as the sum of its marginal probabilities:

$$\hat{\theta}_x = \sum_{y \in \Omega_Y} \hat{\theta}_{x,y} \tag{9}$$

In Step C of Figure 1, for example, the user has negated the selection condition of the *sex* attribute, which translates to the following query:

```sql
SELECT salary, COUNT(*)
FROM census
WHERE sex <> 'Female'
GROUP BY salary
```

In this example, the Result Cache already has frequency estimators (1) $\hat{\theta}_{salary}$ and (2) $\hat{\theta}_{salary|Female}$. We can use the Law of Total Probability to marginalize across the *sex* attribute in order to compute $\hat{\theta}_{salary|\ Female}$, again without actually having to scan any data.

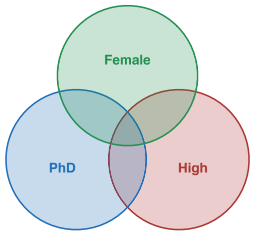
3.2.3 *Inclusion-Exclusion Principle*

Unlike filter chains, where downstream visualizations represent strict subpopulations of upstream visualizations, users can also specify predicates that represent the intersection of selected subpopulations. For example, the top right bar chart in Step D of Figure 1 shows the intersection of the 'Female' and 'PhD' subpopulations, and the bottom right bar chart shows the intersection between non-'Female' and 'PhD' subpopulations, which translates to the following SQL query:

```sql
SELECT salary, COUNT(*)
FROM census
WHERE sex <> 'Female' AND education = 'PhD'
GROUP BY salary
```

In the example, we must compute $\hat{\theta}_{salary,PhD,\neg Female}$. By using the *Inclusion-Exclusion Principle* (IEP) from probability theory, we can rewrite the query to entirely reuse past results stored in the Result Cache without having to scan any tuples in the Sample Store. In order to understand this rewrite rule, consider the rewritten query and the accompanying Venn diagram:

$$\begin{aligned}\hat{\theta}_{High,PhD,\neg Female} = \hat{\theta}_{High} + \hat{\theta}_{PhD} \\ - \hat{\theta}_{High,\neg PhD} \\ - \hat{\theta}_{\neg High,PhD} \\ - \hat{\theta}_{High,PhD,Female}\end{aligned}$$



Each of the terms in the above equation maps to a region of the Venn diagram. For example, the red circle represents $\hat{\theta}_{High}$, and the region of the red circle not overlapping with the blue circle represents $\hat{\theta}_{High,\neg PhD}$. Visually, we can see that $\theta_{High,PhD,\neg Female}$ can be rewritten in many ways, and the above equation is one way to rewrite the query that uses only previously computed estimates from the running example available in the Result Cache.

The IEP is a very powerful rewrite rule and can be applied to a broad range of additional queries by considering the relationship between predicate attributes. For example, if the user switches a Boolean operator (e.g., changing the predicate to `sex<>'Female'` **OR** `education='PhD'`), we can calculate the frequency of the union of two subpopulations simply by reusing our estimate for the intersection.

We can also use the IEP to take advantage of the mutual exclusivity of certain predicates. In particular, if the user applies a predicate representing the intersection of mutually exclusive subpopulations, we can apply the IEP to determine that no tuples can possibly exist in the result, therefore immediately returning a frequency estimate of zero (e.g., a query with predicate `sex='Male'` **AND** `sex='Female'` has a frequency of zero). Similarly, if the user applies a predicate representing the union of mutually exclusive subpopulations, we can again apply the IEP to immediately return a frequency equal to the sum of the subpopulations (e.g., a query with predicate `sex='Male'` **OR** `sex='Female'` has a frequency equal to $\hat{\theta}_{Male} + \hat{\theta}_{Female}$).

3.3 Result Error Propagation

As previously mentioned, users need to know how much faith to place in an approximate query result. Section 2.3 shows that computing the error for a simple query with no selections is relatively straightforward, but, when reusing past results, we need to carefully consider how the (potentially different) errors from each of the individual approximations contributes to the overall error of the new query result. Therefore, we apply well-known propagation of uncertainty principles in order to formally reason about how error propagates across overlapping queries.

For example, consider again the query representing Step B of Figure 1. In this case, we approximate the result of the query as $\hat{\theta}_{salary|Female}$ multiplied by $\hat{\theta}_{Female}$, so we need to consider the error associated with both terms. Note that each of these terms has a different error; that is, the error for the estimate $\hat{\theta}_{Female}$ is lower because it was started earlier (in Step A). After computing the sum of the normalized standard error (Equation 3) across all attribute values (Section 2.3), we combine the error terms for $\hat{\theta}_{salary|Female}$ and $\hat{\theta}_{Female}$ using the propagation of uncertainty formula for multiplication, yielding the estimated error for the final result.

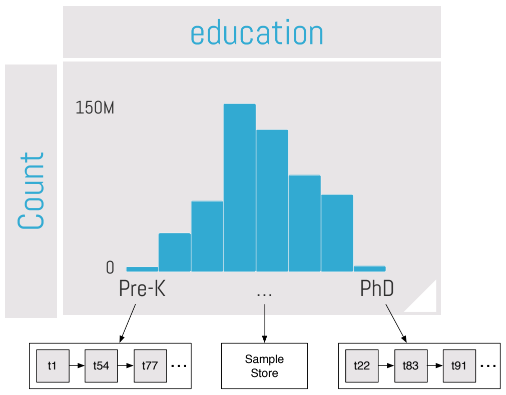


Figure 3: A Tail Index built on *education*.

These error estimates are also useful during the query optimization process. Since queries can often be rewritten into many alternative forms, the Query Engine can pick the series of rewrites that produces an approximate query result with the lowest expected error through a dynamic programming optimization process that recursively enumerates rewritten alternatives. In some cases, it is even possible that leveraging past results might produce an approximate result with higher error than simply computing an estimate by scanning the Sample Store, so the Query Engine must decide if a rewrite is beneficial at all.

4. TAIL INDEXES

Since the data exploration process is user-driven and inherently conversational, the AQP engine is often idle while the user is interpreting a result and thinking about what query to issue next. This "think time" not only allows the system to continue to improve the accuracy of previous query results but also to prepare for future queries. Some existing approaches (e.g., ForeCache [4], DICE [19]) leverage these interaction delays to model user behavior in order to predict future queries, but these techniques typically require a restricted set of operations (e.g., pre-fetching tiles for maps, faceted cube exploration) or a large corpus of training data to create user profiles [28]. Instead, as described in Section 2, our approach leverages a user's "think time" to construct Tail Indexes in the background during query execution to supplement our AQP formulation.

This section first describes how to build a Tail Index on-the-fly based on the most recently issued query in preparation for a potential subsequent query on a related subpopulation. Then, we explain the intricacies of how to safely use these Tail Indexes to support future queries by preserving the randomness properties necessary for our AQP techniques. Finally, we discuss how to extend Tail Indexes to support continuous attributes.

4.1 Building a Tail Index

Many of the query rewrite rules described in Section 3 rely on observing tuples belonging to specific subpopulations (e.g., 'Male'), which will appear frequently in a scan of the Sample Store. However, when considering rare subpopulations (e.g., 'PhD'), tuples belonging to these subpopulations will not be common enough in a scan of the Sample Store to provide low-error approximations within the interactivity threshold. As such, we need to supplement our formulation with indexing techniques in order to provide enough relevant tuples for low-error approximations.

Many existing indexing techniques (e.g., B-trees, sorting, database cracking [14]) organize all tuples without regard for their frequency in the dataset, resulting in unnecessary overhead since our AQP techniques can already provide good approximations over common subpopulations. Furthermore, these indexing techniques destroy the randomness property that our AQP formulation requires, and trying to correct for the newly imposed order would be prohibitively expensive. Therefore, we propose a low-overhead partial index structure [32], called the *Tail Index*, that is built online during a user's exploration session. Tail Indexes dynamically keep track of rare subpopulations to support query rewrites and save space by not indexing tuples with common values, all while also preserving the randomness requirements necessary for our AQP techniques.

Figure 3 shows a sample Tail Index built on the *education* attribute from Step D of Figure 1. The Tail Index is a hash-based data structure that points to either (1) the Sample Store when an attribute value is common or (2) a linked list of pointers to tuples when the attribute value is rare. In the figure, notice that the attribute values in the mass of the distribution (e.g., 'HS') point to the Sample Store, whereas the values in the tails of the distribution (e.g., 'Pre-K', 'PhD') point to lists of indexed tuples.

In order to build this index, though, we need to determine whether a given tuple belongs to a rare subpopulation. Specifically, we decide which attribute values should be indexed by determining if the specified confidence level for a possible future visualization will not be achievable within the interactivity threshold. That is, if the frequency of an attribute value is high enough to provide sufficient tuples from a given subpopulation in order to meet the specified confidence level from a scan of the Sample Store, then the index entry should instead point to the Sample Store. Otherwise, the Tail Index retains the linked list and continues appending new tuples until either the resources need to be reallocated or a maximum index size has been reached.

For longer chains of visualizations with multiple filter conditions, we build an index for each attribute in the chain. For example, if the user has the *sex* visualization linked to *salary*, as in Step B of the example, our techniques would build a multidimensional index on *sex,salary* (i.e., both the 'Male' and the 'Female' buckets point to an index of *salary*, each of which contains only males or females, respectively). However, since longer chains with selections result in increasingly rare subpopulations, the frequency $\hat{\theta}_x$ of the attribute must be adjusted for the value in the entire population. Therefore, if no upstream visualizations are indexed, $\hat{\theta}_x$ is the joint probability (i.e., $\hat{\theta}_{x,y}$). When an attribute value is indexed, subsequent downstream visualizations should instead use the conditional probability (e.g., $\hat{\theta}_{x|y}$) since logically the index defines a new population.

Typically, users cannot keep track of more than a few attributes at a time [7], ensuring that the indexes will not become larger than a few dimensions. Furthermore, the fact that Tail Indexes keep track of only rare subpopulations further reduces their size.

4.2 Using a Tail Index

Recall that the rewrite rules from Section 3 often require tuples that belong to specific subpopulations (e.g., $\hat{\theta}_{salary|Female}$ requires only tuples with a value of 'Female'). For selection operations over a single attribute value (e.g., 'Female'), scanning the indexed tuples avoids the wasted work associated with examining all tuples and ignoring those in which the user has no interest. The rarer a selected subpopulation, the more wasted work the Tail Index will save.

Interestingly, using a Tail Index to answer a query that involves multiple selected values (e.g., selecting both 'Pre-K' and 'PhD' in Step D of Figure 1) is not as straightforward as in a traditional index because of the randomness property that our AQP formulation requires. For example, if a user has selected a subset $\Omega_{Y'}$ of values from Ω_Y, sequentially scanning each list of tuples in full would destroy the randomness, potentially resulting in biased estimates. For example, suppose the user had selected both 'Pre-K' and 'PhD' to filter *salary* in Step D of Figure 1. By scanning each of the subpopulations sequentially (i.e., first scanning all of the 'Pre-K' tuples and then all of the 'PhD' tuples) in order to compute the approximation, the randomness requirement for our AQP formulation is destroyed.

For this reason, we scan each of the buckets in the index proportionally to its frequency to ensure randomness. To determine the list from which to draw the next tuple, we first compute the weight of each attribute value y' in the set of selected values $\Omega_{Y'}$ as:

$$weight(y') = \hat{\theta}_{y'} \sum_{y' \in \Omega_{Y'}} \frac{1}{\hat{\theta}_{y'}} \tag{10}$$

By normalizing the weights of all selected values, each value has a likelihood proportional to its selectivity. Then, the Query Engine can draw from each list based on these weights, ensuring that the samples used to compute the approximation are unbiased.

Finally, for queries over both common and rare subpopuations (e.g., selecting both 'HS' and 'PhD' in Step D of Figure 1), one bucket in the index will point to the base data while the other bucket will point to the list of rare tuples. In this case, we can simply scan the base data and ignore the Tail Index, since the error of the approximation will be dominated by the common value, and the rare value will be observed in the base data at the same rate (relative to the common value) that it would be sampled from the index.

4.3 Indexing Continuous Attributes

As previously mentioned, we can use binning to treat continuous attributes like nominal attributes. However, the user may often be interested in *zooming* in on the distribution within a specific subrange of a continuous attribute, which is a special case of selection where the bounds of a continuous attribute predicate are incrementally narrowed. For example, consider a visualization of the *age* attribute with bins on the decade scale (e.g., 20, 30, 40) where the user has filtered the range between 20 and 40 years old.

Building a complete index (e.g., a B-tree) is both impractical and unnecessary, since users visualize a high-level, aggregated view of the data and do not examine individual tuples. Thus, the simplest way to index a continuous attribute is to treat it as a nominal attribute, where each bin (i.e., predefined subrange) is equivalent to an attribute value. Depending on rarity, bins in the index store either a pointer to the Sample Store or to indexed tuples from that bin's range, similar to a Tail Index for a nominal attribute.

Then, for selections over a set of bins, we can use the techniques described in the previous section for using an index.

However, unlike nominal attributes, zooming presents a new challenge for building Tail Indexes: suppose a common bin contains one or more rare sub-bins at the next zoom level. Then, if the user zooms into one of these sub-bins, we may be unprepared to provide a low-error approximation within the interactivity threshold because the parent bin is not rare and therefore will not be indexed. To try to mitigate this problem, we can transparently divide each bin into sub-bins (e.g., partition *age* into bins of one year instead of ten years), such that if the user performs a zoom action on a rare sub-bin, the system can draw tuples from the index. Since we are always one step ahead of the user, we can use the time between user interactions to continue to compute more fine-grained zoom levels in the event that the user continues zooming on a particular subrange.

5. MORE COMPLEX QUERIES

So far, we have described how our AQP formulation applies in the context of count queries with selection predicates. We now expand our discussion to queries with (1) multiple group-by attributes, (2) aggregate functions other than count, and (3) joins.

5.1 Multiple Group-by Attributes

Rather than changing the selection predicate for a filtered bar chart to understand the relationship between two attributes, users sometimes find it easier to view them in a two-dimensional heatmap, where the gradient of each cell represents the count. For example, a heatmap showing the *sex* attribute plotted against *salary* translates to the following query:

```
SELECT sex, salary, COUNT(*)
FROM census
GROUP BY sex, salary
```

To build a heatmap over two attributes X and Y, we must estimate the joint probability for each cell (i.e., $\hat{\theta}_{x,y}$ for every combination of x and y). We can again use the Chain Rule (Section 3.1) to rewrite $\hat{\theta}_{x,y}$ as either (1) $\hat{\theta}_{x|y}\hat{\theta}_y$ or (2) $\hat{\theta}_{y|x}\hat{\theta}_x$, and then we calculate the error using the previously described error propagation techniques (Section 3.3). As explained, since the Query Engine has a choice between how to rewrite the query, we can automatically pick the rewrite that minimizes error in order to return a better approximation.

Since heatmaps can also act as filters for any linked downstream visualization, we similarly index rare attribute values in order to efficiently answer future queries over selected subpopulations, which in this case are comprised of some combination of (x, y) pairs (i.e., a two-part conjunctive predicate). We can then use the previously described index weighting techniques (Equation 10) to proportionally scan indexes in order to preserve randomness.

5.2 Other Aggregate Functions

Unlike nominal attributes, users can apply other aggregate functions (e.g., average, sum) in addition to counts. In order to compute the estimates and the error for a query involving other aggregate functions, we need to modify our previously described techniques and incorporate properties about these aggregates into our formulation. Furthermore, unlike for a simple count, the error of other aggregate functions for a particular bin is not entirely influenced by the frequency of that bin. Intuitively, observing a tuple with any attribute value will lower the total error for a count visualization by increasing the size of the sample n in the denominator (Equation 1). On the other hand, for an aggregate like average, observing a tuple with a value in a particular bin will only impact the error for the observed bin. For this reason, we need to take special care to compute the error for visualizations that depict average and sum aggregates, including normalizing their errors to the same range as count visualizations.

5.2.1 *Average*

To compute the error of an estimated average for a single bin x of a continuous attribute, we use the sample standard deviation of tuples that belong in that bin. First, we compute the average squared distance of each value in the bin to the bins's mean. Then, we divide the average squared distance by n and take the square root to yield the sample standard deviation.

5.2.2 *Sum*

The error associated with a sum estimate for an attribute value x requires estimates for both the count and average of the attribute value. For this reason, by default, we always estimate average and count for all continuous attributes, as well as the corresponding error estimates. Then, to estimate the error associated with the average, we use the previously described error propagation techniques (Section 3.3) to combine the errors of the average and count estimates.

5.3 Joins

So far, we have focused on the scenario where a user explores a single table or denormalized dataset (e.g., materialized join results, star schema data warehouses). However, extending our AQP formulation to work on joins is useful in order to apply our techniques to more scenarios.

Joins pose several interesting questions, since our AQP techniques require a random sample. Unfortunately, DBMSs usually sort or hash data to compute the result of a join, which breaks our randomness requirement. However, existing techniques (e.g., ripple join [11], SMS join [17], wander join [21]) can be used to provide random samples from the result of a join. Therefore, we can apply our previously described AQP techniques over the result of a join between multiple tables to provide fast and accurate approximations. However, we leave extensions to our AQP formulation that optimize join processing as future work.

6. EVALUATION

We evaluated our techniques using real-world datasets and workloads that were derived from a past user study [7] in order to show that they can significantly improve interactivity and return higher quality answers faster than alternative approaches. First, we compare our prototype IDEA implementation to standard online aggregation, as well as a state-of-the-art column store DBMS that represents the most optimal blocking solution. Then, we show how our techniques perform when scaling different parameters from the benchmark. All experiments were conducted on a single server with an Intel E5-2660 CPU (2.2GHz, 10 cores, 25MB cache) and 256GB RAM.

#1	sex
#2	education
#3	education **WHERE** sex='Female'
#4	education **WHERE** sex='Male'
#5	sex, education
#6	sex **WHERE** education='PhD'
#7	salary
#8	salary **WHERE** education='PhD'
#9	sex, salary
#10	salary **WHERE** sex='Female'
#11	salary
#12	salary **WHERE** sex='Female'
#13	salary **WHERE** sex<>'Female'
#14	salary **WHERE** sex='Female' **AND** education='PhD', salary **WHERE** sex<>'Female' **AND** education='PhD'
#15	age
#16	salary **WHERE** 20<=age<40 **AND** sex='Female' **AND** education='PhD', salary **WHERE** 20<=age<40 **AND** sex<>'Female' **AND** education='PhD'

(a) Census

#1	age
#2	weight
#3	weight **WHERE** age>=16
#4	age
#5	weight
#6	weight **WHERE** age<8
#7	sex
#8	sex **WHERE** age>=16
#9	sex
#10	sex **WHERE** age<8
#11	weight, sex
#12	age **WHERE** weight>=2
#13	age **WHERE** weight>=2 **AND** sex='I'
#14	age **WHERE** weight>=2 **AND** sex<>'I'
#15	age **WHERE** weight<0.4
#16	age **WHERE** weight<0.4 **AND** sex='I'

(b) Abalone

#1	score
#2	abv
#3	abv **WHERE** score>=8
#4	score
#5	abv
#6	abv **WHERE** score<8
#7	so2
#8	so2 **WHERE** score>=8
#9	so2
#10	so2 **WHERE** score<8
#11	so2, abv
#12	score
#13	score **WHERE** abv>=13
#14	score **WHERE** 100<=so2<200 **AND** abv>=13
#15	score **WHERE** abv>=13
#16	score **WHERE** 100<=so2<200

(c) Wine

Figure 4: Exploration Session Definitions

		#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16
Census	MonetDB	0.34	0.39	5.40	8.70	0.48	1.20	1.20	0.91	0.53	4.80	0.42	4.70	1.10	5.60	1.60	7.10
	Online Agg	0.05	0.24	0.78	0.59	0.24	0.46	0.04	0.48	0.07	0.11	0.04	0.11	0.08	7.53	0.29	24.3
	IDEA	0.09	0.29	0.42	0.00	0.00	0.00	0.09	0.12	0.00	0.17	0.00	0.00	0.00	0.48	0.37	2.87
Abalone	MonetDB	0.69	1.30	0.79	0.71	1.30	1.10	0.38	0.42	0.35	0.56	1.30	0.79	4.60	0.90	1.40	7.90
	Online Agg	0.42	0.64	3.17	0.43	0.66	0.71	0.07	0.33	0.07	0.11	0.95	3.93	4.19	4.02	0.79	0.98
	IDEA	0.48	0.71	0.82	0.00	0.00	0.33	0.13	0.09	0.00	0.05	0.00	0.37	0.42	0.00	0.34	0.39
Wine	MonetDB	0.75	0.90	0.90	0.75	0.89	1.60	1.30	0.87	1.30	2.10	1.30	0.69	0.85	1.40	0.84	1.20
	Online Agg	0.14	0.16	1.36	0.13	0.16	0.28	0.11	0.40	0.10	0.13	0.19	0.13	1.07	1.56	1.06	0.22
	IDEA	0.25	0.24	0.39	0.00	0.00	0.00	0.18	0.16	0.00	0.00	0.00	0.00	0.26	0.20	0.00	0.27

Figure 5: Benchmark Runtimes (s)

6.1 Setup

We compare our prototype IDEA system against a baseline of standard online aggregation (also implemented in our prototype), as well as a column store DBMS (MonetDB 5) using three data exploration sessions that were derived from traces taken from a past user study [7]. Although MonetDB does not compute approximate answers, we wanted to show how the presented AQP techniques compare to a state-of-the-art system based on the traditional blocking query execution paradigm.

To construct each exploration session, we identified the most common sequences that users performed and then synthesized them into 16 distinct queries for each of the three datasets [22] (Census, Abalone, Wine). These exploration sessions model how a user would analyze a new dataset in order to uncover new insights, typically starting out with broader queries that are iteratively refined until arriving at a more specific conclusion. For example, the goal of the Census session is to determine which factors (e.g., *sex*, *education*) influence whether an individual falls into the 'High' or 'Low' annual *salary* bracket. The Abalone session explores data about a type of sea snail to identify which characteristics (e.g., *weight*) are predictive of *age*, which is generally difficult to determine. Finally, the Wine session examines the relationship between chemical properties (e.g., *so2*) and a wine's quality *score*.

To standardize all of the exploration sessions, we scaled each of the datasets to $500M$ tuples while preserving the original distributions of the attribute values. We additionally parse and load all datasets into memory for all systems before executing the benchmarks.

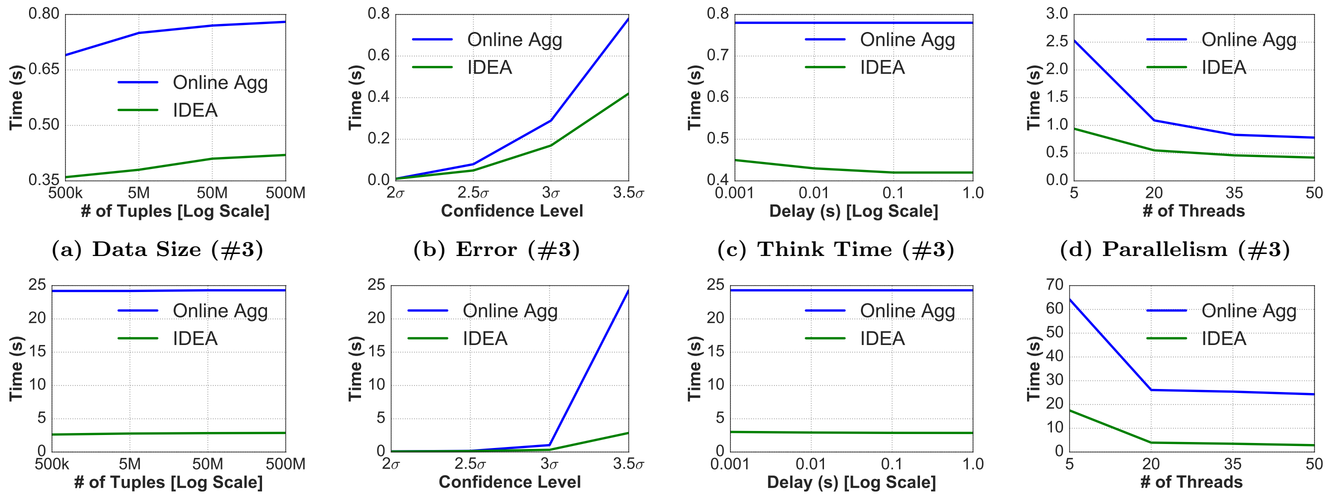
Since both online aggregation and our IDEA prototype can provide approximate results, we set the confidence level for query results to 3.5σ (i.e., the expected deviation of our approximation from the true result is less than 0.05%) before moving on to the query for producing the next visualization in the exploration session. Based on the interaction logs from our user study, we wait one second after achieving a sufficiently low result error before issuing the next query in order to model the time taken by the user to interpret the displayed results and decide the next action to perform (i.e., "think time"). Finally, we use 50 threads per query operation, which maximizes the runtime performance on the given hardware.

In Section 6.3, we vary each of these parameters (i.e., data size, error threshold, "think time", and amount of parallelism) to show how our techniques scale compared to online aggregation.

6.2 Overall Performance

Figure 5 shows the time (in seconds) required to return an answer at or above the specified confidence interval in (1) MonetDB, (2) Online Aggregation, and (3) our IDEA prototype for every step in each of the simulated exploration sessions. Again, as a system with a blocking query execution model, MonetDB must wait until computing the exact query result (i.e., returning an answer with 100% confidence).

Each step number in the session corresponds to an action that the user performs in the visual interface, and Figure 4 shows the corresponding queries executed to produce each visualization. Since all queries compute the count grouped by an attribute, we show only the group-by attribute followed by any selection predicates. For example, in the Cen-



(a) Data Size (#3) (b) Error (#3) (c) Think Time (#3) (d) Parallelism (#3)

(e) Data Size (#16) (f) Error (#16) (g) Think Time (#16) (h) Parallelism (#16)

Figure 6: Scalability Microbenchmarks

sus exploration session, the simulated user first visualizes both *sex* (#1) and *education* (#2), then views the distribution of *education* for only the 'Female' subpopulation (#3). Sometimes, an action in the frontend (e.g., removing a link) will issue multiple concurrent queries, in which case we delimit individual queries with a comma (e.g., #5 and #16 in Census). In these cases, we report the runtime after the results for both queries reach the acceptable confidence level, since the simulated user cannot proceed without low-error approximations for all visualizations on a given step. Since the main goal of our work is to provide a high-quality result within the interactivity threshold (i.e., 500ms), each cell in Figure 5 is color-coded to indicate how close the runtime is to the threshold, ranging from green (i.e., significantly below the threshold) to red (i.e., above the threshold).

As shown in Figure 5, our IDEA prototype can return high-quality approximations within the interactivity threshold in many cases where both MonetDB and online aggregation cannot. In some cases, however, MonetDB can return an exact result faster than online aggregation can return an approximate answer with an acceptable error, due to a number of optimizations (e.g., sorting, compression) performed at data load time. Unfortunately, in interactive data exploration settings, the extensive preprocessing necessary to perform these types of optimizations often represents a prohibitive burden for the user.

Although our techniques cannot always guarantee a result with acceptable error in less than 500ms (e.g., #16 in Census, #2 and #3 in Abalone), our IDEA prototype returns high-quality approximate results as interactively as possible. Our approach is slightly slower than online aggregation for some queries that do not benefit from any of our proposed techniques because of the minor additional overhead associated with result caching and index construction. Note, however, that this overhead is small enough that it never causes our techniques to exceed the interactivity threshold where online aggregation does not already.

We now discuss the performance of our IDEA prototype to the online aggregation baseline in more detail for each of the previously described exploration sessions.

6.2.1 Census

After viewing the distribution of *education* for only the 'Female' subpopulation (#3), the simulated user changes the selection to examine *education* for the 'Male' subpopulation instead (#4). In this case, our prototype rewrites the query using the Law of Total Probability to reuse previously computed results and return a result without having to scan any data from the Sample Store. Similarly, when the simulated user reverses the direction of the link between *sex* and *education* to view the distribution of *sex* for only individuals with a 'PhD' (#6), our prototype uses Bayes' theorem to rewrite the query to return an answer almost immediately. When the simulated user issues two queries to compare the salary for male and female doctorates (#14), we can first execute the much easier (i.e., less selective) query over the non-'Female' subpopulation and use the Inclusion-Exclusion Principal to rewrite the query in order to compute an approximation of *salary* for only the female doctorates. To execute this easier query, we use the multidimensional Tail Index created over the *education* and *sex* attributes. Finally, when the user drills down into especially rare subpopulations (#14 and #16), our Tail Indexes allow our IDEA prototype to compute a low-error approximation significantly faster than online aggregation.

6.2.2 Abalone

The simulated user first views the distribution of *age* (#1) and *weight* (#2) individually, then compares the distributions of *weight* for older (#3) and younger (#6) abalones. Since both of these selected subpopulations are relatively rare, our Tail Indexes help us to achieve a low-error result significantly faster than online aggregation. Our Tail Indexes provide similar speedups in steps #13 and #15, yet again achieving interactivity where online aggregation cannot. Finally, when the simulated user compares *age* for infant abalones versus adults (#14), we can leverage the Law of Total Probability to reuse previously computed answers from the Result Cache in order to return an approximation instantly.

6.2.3 Wine

As part of this exploration path, the simulated user compares the alcohol content *abv* of high quality wines (#3) to that of lesser quality wines (#6). Although online aggregation cannot provide a sufficiently low-error approximation for the query over only the high quality wines (a rare subpopulation), our Tail Indexes allow us to compute a high-quality approximation within the interactivity threshold. For the query over the lesser quality wines, we can use the Law of Total probability to reuse the results from #2 and #3 to provide a low-error approximation instantly. Finally, by caching the results of previously executed queries, we can immediately return the results for the queries that the simulated user has already issued in the past (#4, #5, #9, #11, #12, #15).

6.3 Scalability

To better understand the limits of our techniques, we conducted scalability experiments in which we varied (1) data size, (2) confidence level, (3) simulated "think time", and (4) amount of parallelism. For all scalability experiments, we show the results for steps #3 and #16 from the Census workload, which represent an early and late stage in the exploration session, respectively. As such, step #3 has much less potential for result reuse and indexing than #16, which has access to cached results from past queries and indexes that have already been created.

6.3.1 Data Size

Figures 6a and 6e show the runtimes for step #3 and #16, respectively, for various data sizes with a fixed confidence level of 3.5σ, "think time" of one second, and parallelism of 50 threads. As shown, for both #3 and #16, the size of the data has little noticeable impact on performance, since both online aggregation and our IDEA prototype scan only a fraction of the tuples (independent of data size) in order to provide a low-error approximation. The slight constant increase in the runtime for both approaches as data size increases is primarily attributable to the overhead associated with managing larger datasets in memory (e.g., object allocation, garbage collection, CPU caching effects), which increase with the amount of data.

6.3.2 Error

To show how both online aggregation and our proposed AQP techniques perform for different amounts of acceptable error, we show the runtime for both online aggregation and our IDEA prototype to achieve varying confidence levels for step #3 and #16 in the Census exploration session using $500M$ tuples, one second of "think time," and 50 threads per query operation. As shown in Figures 6b and 6f, our approach is able to provide an approximate answer with the same amount of error as online aggregation in less time. In some cases, such as with a confidence of 3.5σ, our approach outperforms online aggregation by an order of magnitude due to our previously described Tail Indexes.

6.3.3 Think Time

Since the amount of time between user interactions influences the number of tuples that can be indexed, Figures 6c and 6g show the time necessary for both online aggregation and our IDEA prototype to compute an approximate result with a confidence level of 3.5σ using $500M$ tuples and 50 threads for each query operation. As shown, the performance of online aggregation is only very slightly affected by "think time". On the other hand, as shown, the more "think time" that our IDEA prototype has to begin preparing for a follow up query, the faster it can compute a low-error visualization.

6.3.4 Parallelism

As previously described, each query operation (e.g., index creation, aggregate approximation) uses multiple threads in order to best take advantage of modern multi-core systems. To show how the amount of parallelism affects system performance, Figures 6d and 6h show the runtime to compute an approximate result with a confidence level of 3.5σ for both online aggregation and our IDEA prototype using $500M$ tuples, and one second of "think time" for a varying number of threads. As shown, as the number of threads increases, the system can take better advantage of the underlying hardware, resulting in lower query latencies. However, for more than 50 threads, performance begins to degrade due to thrashing and increased contention on shared data structures.

7. RELATED WORK

The techniques presented in this work have overlap in three main areas: (1) approximate query processing, (2) result reuse, and (3) indexing.

7.1 Approximate Query Processing

In general, AQP techniques fall into two main categories: (1) biased sampling [1] and (2) online aggregation [13]. Systems that use biased sampling (e.g., AQUA [2], BlinkDB [3], DICE [19]) typically require extensive preprocessing or foreknowledge about the expected workload, which goes against the ad hoc nature of interactive data exploration. Similarly, systems that perform online aggregation (e.g., CONTROL [12], DBO [18], HOP [5], FluoDB [35]) are unable to give good approximations in the tails of the distribution, which typically contain the most valuable insights. Our approach, on the other hand, leverages unique properties of interactive data exploration in order to provide low-error approximate results without any preprocessing or a priori knowledge of the workload.

Similar to our approach, Verdict [29] uses the results of past queries to improve approximations for future queries in a process called "Database Learning." However, Verdict requires upfront offline parameter learning, as well as a sufficient number of training queries in order to begin seeing large benefits.

7.2 Result Reuse

In order to better support user sessions in DBMSs, various techniques have been developed to reuse results [33, 16, 26, 9]. These techniques, however, do not consider reuse in the context of (partial) query results with associated error. Specifically, our proposed AQP formulation allows us to formally reason about error propagation to quantify result uncertainty for the user, as well as for making query optimization decisions.

7.3 Indexing

Database cracking [14] is a self-organizing technique that physically reorders tuples in order to more efficiently support selection queries without needing to store secondary indexes. However, database cracking would destroy the randomness of the underlying data, a property that our AQP techniques rely upon in order to ensure the correctness of approximate results.

Partial indexes [32] are built online for only the subset of data of interest to the user. Our Tail Index takes these ideas a step further and attempts to keep track of only rare subpopulations in order to minimize overhead while still meeting interactivity requirements. Moreover, our techniques necessitate careful consideration of how to draw tuples from Tail Indexes in order to preserve randomness.

8. CONCLUSION

In this paper, we presented a novel AQP formulation that treats query results as random variables in order to provide low-error approximate results at interactive speeds, even for queries over rare subpopulations. Our formulation leverages the unique properties of interactive data exploration, specifically opportunities for result reuse across queries and user "think time." We proposed several optimization techniques that are based on probability theory, including new query rewrite rules and index structures. We implemented these techniques in a prototype system and showed that they can achieve interactive latency in many cases where alternative approaches cannot.

9. ACKNOWLEDGMENTS

This research is funded in part by the NSF CAREER Award IIS-1453171, NSF Award IIS-1514491, Air Force YIP AWARD FA9550-15-1-0144, and the Intel Science and Technology Center for Big Data, as well as gifts from Google, VMware, Mellanox, and Oracle.

10. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-By Queries. In *SIGMOD*, pages 487–498, 2000.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *SIGMOD*, pages 574–576, 1999.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, pages 29–42, 2013.

[4] L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD*, pages 1363–1375, 2016.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, pages 313–328, 2010.

[6] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive Analytics through Pen and Touch. In *VLDB*, pages 2024–2027, 2015.

[7] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The Case for Interactive Data Exploration Accelerators (IDEAs). In *HILDA*, 2016.

[8] C. Dunne, N. H. Riche, B. Lee, R. A. Metoyer, and G. G. Robertson. GraphTrail: Analyzing Large Multivariate, Heterogeneous Networks while Supporting Exploration History. In *CHI*, pages 1663–1672, 2012.

[9] K. Dursun, C. Binnig, U. Çetintemel, and T. Kraska. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*, pages 1275–1289, 2017.

[10] 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 Min. Knowl. Discov.*, pages 29–53, 1997.

[11] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, pages 287–298, 1999.

[12] 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. *IEEE Computer*, pages 51–59, 1999.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD*, pages 171–182, 1997.

[14] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[15] Y. E. Ioannidis and S. Viglas. Conversational Querying. *Inf. Syst.*, pages 33–56, 2006.

[16] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An Architecture for Recycling Intermediates in a Column-Store. In *SIGMOD*, pages 309–320, 2009.

[17] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A Disk-Based Join With Probabilistic Guarantees. In *SIGMOD*, pages 563–574, 2005.

[18] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. In *SIGMOD*, pages 725–736, 2007.

[19] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and Interactive Cube Exploration. In *ICDE*, pages 472–483, 2014.

[20] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid Sampling for Visualizations with Ordering Guarantees. In *VLDB*, pages 521–532, 2015.

[21] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*, pages 615–629, 2016.

[22] M. Lichman. UCI Machine Learning Repository, 2013.

[23] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *TVCG*, pages 2456–2465, 2013.

[24] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *TVCG*, pages 2122–2131, 2014.

[25] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time Visual Querying of Big Data. *Comput. Graph. Forum*, pages 421–430, 2013.

[26] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in Pipelined Query Evaluation. In *ICDE*, pages 338–349, 2013.

[27] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[28] A. Ottley, H. Yang, and R. Chang. Personality as a Predictor of User Strategy: How Locus of Control Affects Search Strategies on Tree Visualizations. In *CHI*, pages 3251–3254, 2015.

[29] Y. Park, A. S. Tajik, M. J. Cafarella, and B. Mozafari. Database Learning: Toward a Database that Becomes Smarter Every Time. In *SIGMOD*, pages 587–602, 2017.

[30] T. Petty and the Heartbreakers. The Waiting, 1981.

[31] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *TVCG*, pages 52–65, 2002.

[32] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, pages 4–11, 1989.

[33] K. Tan, S. Goh, and B. C. Ooi. Cache-on-Demand: Recycling with Certainty. In *ICDE*, pages 633–640, 2001.

[34] Y. Wu, J. M. Hellerstein, and E. Wu. A DeVIL-ish Approach to Inconsistency in Interactive Visualizations. In *HILDA*, 2016.

[35] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *SIGMOD*, pages 913–918, 2015.

[36] E. Zgraggen, R. C. Zeleznik, and S. M. Drucker. PanoramicData: Data Analysis through Pen & Touch. *TVCG*, pages 2112–2121, 2014.