One Size Does Not Fit All: A Bandit-Based Sampler Combination Framework with Theoretical Guarantees

Jinglin Peng[†] Bolin Ding[◇] Jiannan Wang[†] Kai Zeng[◇] Jingren Zhou[◇]
Simon Fraser University[†] Alibaba Group[◇]
{jinglin_peng, jnwang}@sfu.ca[†] {bolin.ding, zengkai.zk, jingren.zhou}@alibaba-inc.com[◇]

ABSTRACT

Sample-based estimation, which uses a sample to estimate population parameters (e.g., SUM, COUNT, and AVG), has various applications in database systems. A sampler defines how samples are drawn from a population. Various samplers have been proposed (e.g., uniform sampler, stratified sampler, and measure-biased sampler), since there is no single sampler that works well in all cases. To overcome the "one size does not fit all" challenge, we study how to combine multiple samplers to estimate population parameters, and propose SamComb, a novel bandit-based sampler combination framework. Given a set of samplers, a budget, and a population parameter, SamComb can automatically decide how much budget should be allocated to each sampler so that the combined estimation achieves the highest accuracy. We model this sampler combination problem as a multi-armed bandit (MAB) problem and propose effective approaches to balance the exploration and exploitation trade-off in a principled way. We provide theoretical guarantees for our approaches and conduct extensive experiments on both synthetic and real datasets. The results show that there is a strong need to combine multiple samplers, in order to obtain accurate estimations without the knowledge about population predicates and distributions, and SamComb is an effective framework to achieve this goal.

CCS CONCEPTS

• **Information systems → Database query processing**; **Online analytical processing engines**.

KEYWORDS

approximate query processing, sample combination, bandit

ACM Reference Format:
Jinglin Peng, Bolin Ding, Jiannan Wang, Kai Zeng, and Jingren Zhou. 2022. One Size Does Not Fit All: A Bandit-Based Sampler Combination Framework with Theoretical Guarantees. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517900

Acknowledgements. This work was supported in part by Mitacs through an Accelerate Grant, NSERC through a discovery grant and a CRD grant. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517900

1 INTRODUCTION

Sample-based estimation is a fundamental tool in statistics [38, 39]. It uses a (randomly-drawn) sample to estimate population parameters (e.g., SUM, COUNT, and AVG). Since the estimation is derived from a sample whose size is usually small, it is much faster than computing the exact parameters over the entire data. Due to this reason, sample-based estimation has various applications, such as online aggregation [12, 17, 32], approximate query processing [3, 13–15, 23, 28, 29, 31, 34–36, 44, 45], cardinality estimation [11, 18, 24, 27], and exploratory data analysis [33, 43].

One size does not fit all. The estimation accuracy is highly dependent on two important factors simultaneously (which will be elaborated in Section 2.1): i) how data is sampled, i.e., a *sampling distribution* which specifies how likely a tuple is drawn into the sample, and ii) the underlying data distribution in the population where we want to calculate parameters. Various samplers are proposed in the literature [6, 13, 20, 21] with different sampling distributions. For example, a uniform sampler draws a sample from the population where each tuple has the same probability to be selected [17], a stratified sampler draws a sample so that the tuples in each group have the same probability to appear in the sample [6, 21], and a measure-biased sampler draws a sample so that the tuples with large measure values have a higher probability to be selected [13]. While the sampling distribution has to be specified before the sample is drawn, ii) is decided at the "estimation time".

```
SELECT SUM(A) FROM table WHERE B > 10
```

For example, the above query specifies that we want to focus on the population with `B > 10` and estimate `SUM(A)`. Thus, the distribution of `A` in that particular population and how well it matches the sampling distribution decide the estimation accuracy. Therefore, there is no single *sampler* that works well in all cases.

In this paper, we investigate how to combine multiple samplers of the same population to estimate population parameters. Given k different samplers, a *sample budget*, an *aggregation function* (i.e., COUNT, SUM, AVG), and a *predicate* specifying the population to be focused on (e.g., `B > 10`), we study how to allocate the budget to each sampler, so that the combined estimation using k samplers achieves the highest accuracy. For ease of presentation, we use SQL queries (e.g., in the example above) to represent population parameters to be estimated. Similar to existing work [4, 13], we assume each sampler pre-computes a large sample, thus they can efficiently draw a sample with given budget by sequentially scanning the data.

Exploitation vs Exploration. We prove that the optimal budget allocation strategy is to allocate all the budget to the *best* sampler. The next questions is how to identify the best sampler. One may design some heuristic rules by leveraging the query information. Unfortunately, it is insufficient to identify the best sampler by only looking into the query, due the following two reasons.

Firstly, a query could satisfy multiples rules, and resulting in multiple samplers chosen by different rules. For example, consider the following two rules: 1) Rule 1: If a query contains `A` in the aggregate function, then choose the measure-biased sampler over `A` column. 2) Rule 2: If a query contains `B = b` in the predicate, where `b` is a constant, then choose a stratified sampler over `B` column. Consider a query:

```
SELECT SUM(price * (1 - tax)) WHERE country = 'USA'
```

The measure-biased sampler over `price` and `tax` column, and the stratified sampler over `country` both satisfy the rules. Hence it is still unknown which sampler should be chosen.

Secondly, even we have a rule-based approach that returns a single sampler, the best sampler still can be different for different data. For example, consider a query:

```
SELECT SUM(price) FROM table WHERE itemid < 100
```

If all tuples satisfy `itemid < 100`, then the measure-biased sampler over `price` column is known as the best sampler. However, if all the tuples whose `itemid < 100` have a very small price, then the measure-biased sampler could perform worse than a uniform sampler, since the tuples that contributes to the answer most are those with small price, and are assigned with a small probability by the sampler.

The above two reasons suggests that it is unknown which sampler is the best w.r.t. a given query before scanning the whole data. To address this issue, we allocate a certain amount of sampling budget to each sampler (i.e., draw samples of certain size using each sampler) in an adaptive way, and assess which sampler is the best using their samples. The identified one is called the *empirically best sampler*, which may or may not be the *truly best sampler*. During this process, we need to balance the *exploitation vs exploration* trade-off, that is, to exploit the the empirically best sampler or explore a different sampler which could be the truly best sampler.

SamComb Framework. To solve this challenge, we propose SamComb, a bandit-based sampler combination framework. It models sampler combination problem as multi-armed bandit (MAB) problem [5, 26, 40], which is a principled way to balance the *exploitation vs exploration* trade-off. SamComb is an iterative framework. At each iteration, it assesses which sampler is the best, then uses an MAB-based approach to decide which sampler to select, and finally allocates a small budget to the sampler. The iteration will continue until the budget is exhausted. Finally, SamComb uses the budget allocated to each sampler to estimate the answer to the query, respectively, and computes a weighted average of these estimated answers as the final answer. Comparing to approaches that select one best sample, the select-then-combine framework of SamComb can fully leverage all the samples and does not waste any budget used for exploration.

The approaches developed for MAB have a guarantee that most budget will be allocated to the best arm. However, it is not clear whether they are still applicable to our problem, since the two problems have different optimization objectives. MAB aims to maximize a linear sum of the reward from each arm and it only involves sample size (i.e., pull times) as variable, while our problem aims to minimize a complex function of the variance from each sampler: it is not linearly additive as MAB, and it involves both sample size and weight. We propose effective solutions and justify our solutions both analytically and empirically.

Full Table

Id	Price	Country
1	150	USA
2	10	CA
3	0.1	USA
4	350	USA
5	50	CA
6	3	USA
7	650	CA
8	0.5	USA
9	900	USA
10	85	USA
...	...	...
1M	0.2	CA

A sample drawn by uniform sampler

Id	Price	Country	Probability
1	150	USA	1e-6
3	0.1	USA	1e-6
...	...	...	...

A sample drawn by measured-biased

Id	Price	Country	Probability
9	900	USA	0.018
7	650	CA	0.013
...	...	...	...

A sample drawn by stratified sampler

Id	Price	Country	Probability
2	10	CA	2.5e-6
4	350	USA	6.25e-7
...	...	...	...

Figure 1: An illustration of different samplers.

Firstly, we discuss how to model our problem as an MAB problem and then justify this modeling by showing that their objective functions share some common properties.

Secondly, to solve our budget allocation problem, we extend two well-known MAB approaches [5], ϵ_t-greedy and Upper Confidence Bound (UCB). We prove that both approaches can guarantee that the allocated budget to each sub-optimal sampler is at most $O(\ln n)$, where n is the total budget.

Thirdly, we discuss how to combine the estimation from each sampler into the final estimation. We propose two approaches and prove the optimality of both approaches.

Finally, we conduct extensive experiments to evaluate SamComb using both synthetic and real-world datasets. The results validate the effectiveness of our approaches empirically. We also apply SamComb to applications like selectivity estimation, and demonstrate the advantages of combining multiple samplers. We see this work as an initial step towards building a principled federated framework to address the one-size-does-not-fit-all challenge in online aggregation, approximate query processing, and cardinality estimation.

In summary, we make the following contributions:

(1) We propose a novel idea that combines different samplers to enhance sample-based estimation. We formally define the sampler combination problem. To the best of our knowledge, we are the first to study this problem.
(2) We model our problem as a multi-armed bandit problem, and propose SamComb, a bandit-based sampler combination framework to solve the problem.
(3) We propose two budget allocation approaches based on existing MAB strategies, and two weight allocation approaches to combine the estimated answers from multiple samplers into the final answer. We prove theoretical guarantees for these approaches.
(4) We evaluate SamComb on both synthetic and real-world datasets. The results show that SamComb can effectively combine multiple samplers and achieve a higher estimation accuracy compared to baselines.

2 PROBLEM FORMALIZATION

In this section, we first define the concept of sampler and then formalize the sampler combination problem.

We focus on the population parameters that can be expressed in the following form of SQL queries:

```
SELECT f(A) FROM table
WHERE Predicate (B1, B2, ...)
```

where f is SUM, COUNT, or AVG. For ease of presentation, we mainly consider f = SUM, since COUNT is a special case of SUM and AVG is the ratio of SUM to COUNT. We also discuss how to support other aggregate functions such as PERCENTILE in Section 6.

2.1 Sampler

There are different sampling methods (uniform, measure-biased, and stratified sampling). We find that each of them can be seen as a special case of weighted sampling. Thus, we define sampler as weighted sampling. Given a table T with N tuples, a sampler draws a weighted sample from T such that the probability of each tuple being selected is proportional to its weight. Definition 1 presents a formal definition.

Definition 1 (Sampler). *Given a table T with N tuples, each tuple $t_i \in T$ is associated with a probability p_i, where $0 < p_i \leq 1$ and $\sum_{i=1}^{N} p_i = 1$. A sampler S can draw a weighted sample S with replacement of any given sample size from T, where each tuple $t_i \in T$ has a probability of p_i to be sampled.*

If the context is clear, we represent a sampler by its probability of selecting each tuple, i.e., $\{p_1; p_2; \cdots; p_N\}$. The only difference among different sampling methods is how to compute p_i (for all $i \in [1; N]$). Note that one can certainly obtain other samplers by computing p_i differently. Our system supports them as well.

- *Uniform Sampler:* A uniform sampler selects each tuple with the same probability, i.e., $p_i = \frac{1}{N}$.
- *Measure-biased Sampler:* Given a measure column, a measure-biased sampler selects each tuple with probability proportional to the measure value, i.e., $p_i = \frac{m_i}{\sum_{i=1}^{N} m_i}$, where m_i is the measure value for tuple i.
- *Stratified Sampler:* Given a stratified column G, the table is divided into $|\mathrm{dom}(G)|$ groups, where $|\mathrm{dom}(G)|$ is the domain size of G (i.e., the number of distinct values in G). A stratified sampler selects each group with equal probability. For each tuple i, let G_i be the group that contains tuple i, then we have $p_i = \frac{1}{|\mathrm{dom}(G)||G_i|}$.

Example 1 illustrates how each sampler computes p_i.

Example 1. *Consider the example in Figure 1. The full table has 1M tuples. A uniform sampler selects each tuple with equal probability $\frac{1}{1M}$ = 1e-6. Now, consider a measure-biased sampler on Price column. Suppose the total measure of price is $150 + 10 + \cdots + 0{:}2 = 50000$, then the probability of selecting tuple t is computed as $\frac{t.price}{50000}$. E.g., the probability of selecting tuple 9 is $\frac{900}{50000} = 0{:}018$. Finally we consider a stratified sampler on Country column. It has 2 strata: USA and CA, where USA has $0{:}8M$ tuples and CA has $0{:}2M$ tuples. Hence, for the tuples whose country is USA we have $p_i = \frac{1}{2*0{:}8M}$ = 6.25e-7, and for the tuples whose country is CA we have $p_i = \frac{1}{2*0{:}2M}$ = 2.5e-6.*

Answer Estimation. Given a query q and a weighted sample S, let $q(S)$ denote the estimated answer based on S. Hansen-Hurwitz (HH) estimator [16] can be directly applied to estimate the answer to an SUM query when the query has no predicate (e.g., q: SELECT SUM(Price) FROM table). See Equation (1) below.

$$q(S) = \frac{1}{|S|} \sum_{i=1}^{|S|} \frac{y_i}{p_i}; \tag{1}$$

where y_i is the aggregate value (e.g., $t_i[Price]$) of tuple i.

The HH estimator can be easily extended to support predicates through query rewriting. The main idea is to rewrite a with-predicate query as an equivalent without-predicate query. E.g., query SELECT SUM(Price) From table WHERE Predicate can be rewritten as: SELECT SUM(CASE WHEN Predicate THEN Price ELSE 0 END) FROM table. Then, we generalize the $\frac{y_i}{p_i}$ to d_i: $d_i = 0$ if the tuple does not satisfy the predicate, otherwise $d_i = \frac{y_i}{p_i}$.

Estimation Quality. A common approach to measure the estimation quality are confidence intervals, which bound the real result with high probability. To compute the confidence interval, we first define *sampler quality*.

Definition 2 (Sampler Quality). *Given a sampler $\{p_1; p_2; \cdots; p_N\}$ and a SUM query, we define a discrete distribution $\mathcal{D}^q$ w.r.t. query q (abbreviated as $\mathcal{D}$ if the context is clear) that takes value d_i with probability p_i for each $i \in [1; N]$. Define sampler quality by $var(\mathcal{D})$, which is computed as*

$$var(\mathcal{D}) = \sum_{i=1}^{N} p_i \cdot \left(d_i - mean(\mathcal{D})\right)^2; \tag{2}$$

where $mean(\mathcal{D}) = \sum_{i=1}^{N} p_i \cdot d_i$.

Intuitively, the sampler quality measures how good a sampler is for answering a given query. It is the variance of the HH estimator with a sample of 1 tuple drawn from the sampler. Hence, after drawing the same size of samples, the sampler with a higher quality (lower variance) will result in a better estimator.

As each tuple is drawn independently, the variance of the estimator over a sample of size S can be expressed as $\frac{\mathrm{var}(\mathcal{D})}{|S|}$. Based on the central limit theorem (CLT), the confidence interval is computed as

$$CI = q(S) \pm \lambda \sqrt{\frac{\mathrm{var}(\mathcal{D})}{|S|}} \tag{3}$$

where λ is a constant number related to the confidence level. For example, $\lambda = 1{:}96$ means the true value lies in the confidence interval with the probability of 95%.

The larger the width of the confidence interval (i.e., $\lambda \sqrt{\frac{\mathrm{var}(\mathcal{D})}{|S|}}$), the lower quality the estimated answer. Given a query, we prefer a sampler with $\mathrm{var}(\mathcal{D})$ as small as possible. Thus, $\mathrm{var}(\mathcal{D})$ is named *sampler quality* in Definition 2.

For simplicity, let $\mathrm{var}(q(S)) = \frac{\mathrm{var}(\mathcal{D})}{|S|}$ be the variance of $q(S)$. Then, the confidence interval can also be denoted by

$$CI = q(S) \pm \lambda \sqrt{\mathrm{var}(q(S))} \tag{4}$$

Sampler Implementation. Each sampler exposes an interface that takes a sample budget n_i as input and returns a sample of size n_i. To save sampling time, we assume that each sampler has

precomputed a large sample stored on disk in the offline stage. This is achieved by applying the approach in Appendix B of Sample + Seek [13]. During query time, it sequentially scans n_i tuples from the precomputed sample to get a sample of size n_i, or directly computes the aggregate on a sample by issuing a range query with predicate `row_id BETWEEN a AND b`.

2.2 Sampler Combination Problem

Problem Definition. Suppose the total sample budget is n. The sampler combination problem is to study how to allocate the total budget to each sampler such that the *combined estimator* performs the best. Specifically, given a query q, we draw a sample S_i with size n_i from sampler S_i, such that $\sum_{i=1}^{k} n_i = n$. For each sample S_i, we can get an unbiased estimator of the query result, i.e., $q(S_i)$. Then we combine k estimators into the final estimator. Let ψ be the set of drawn samples, i.e., $\psi = \{S_1; S_2; \cdots; S_k\}$. We denote their combined estimator as $q(\psi)$, which is computed as follows:

$$q(\psi) = \sum_{i=1}^{k} w_i \cdot q(S_i); \tag{5}$$

where w_i is the weight for each estimator $q(S_i)$ and is constrained by $\sum_{i=1}^{k} w_i = 1$. Note that we may allocate no budget to a sampler. In this case we remove its sample from ψ.

The variance of the combined estimator is computed as [1]:

$$\text{var}\ q(\psi) = \sum_{i=1}^{k} w_i^2 \cdot \text{var}(q(S_i)) = \sum_{i=1}^{k} w_i^2 \cdot \frac{\text{var}(\mathcal{D}_i)}{n_i} \tag{6}$$

Recall that our goal is to minimize the estimation error (confidence interval) using the combined estimator. Since the confidence interval can be computed from variance (see Section 2.1), our goal is equivalent to minimize the variance of the combined estimator.

We call this problem the sampler combination problem, as formalized in Problem 1.

Problem 1 (Sampler Combination). *Given a set of k samplers, a query q, and a total budget n, the goal of the sampler combination problem is to decide the sample size n_i for each sampler such that the combined estimator has the minimized variance:*

$$\begin{aligned} \underset{n_1;\cdots;n_k;w_1;\cdots;w_k}{arg\,min} \quad & \sum_{i=1}^{k} w_i^2 \cdot \frac{var(\mathcal{D}_i)}{n_i} \\ \textit{subject to} \quad & \sum_{i=1}^{k} n_i = n \\ & n_i \geq 0;\ \textit{for all}\ i \in [1;k] \\ & \sum_{i=1}^{k} w_i = 1 \end{aligned} \tag{7}$$

3 SAMPLER COMBINATION FRAMEWORK

In this section, we discuss how to solve the sampler combination problem and present the sampler combination framework.

[1]Note that each sampler draws sample independently.

3.1 Optimal Weight Allocation

The sampler combination problem consists of two sub-problems:

(1) Budget Allocation. How should we decide the sample size n_i for each sampler?
(2) Weight Allocation. How should we decide the weight w_i for each estimator?

Let us first consider the second problem by assuming $n_1; n_2; \cdots; n_k$ have been decided. The problem can be formalized as follows:

$$\begin{aligned} \underset{w_1;w_2;\cdots;w_k}{arg\,min} \quad & \sum_{i=1}^{k} w_i^2 \frac{\text{var}(\mathcal{D}_i)}{n_i} \\ \text{subject to} \quad & \sum_{i=1}^{k} w_i = 1; \end{aligned}$$

where $n_1; n_2; \cdots; n_k$ are constant values.

By applying Lagrange's method of multipliers[7], we can get the optimal weight allocation: $w_i = \frac{\frac{n_i}{\text{var}(\mathcal{D}_i)}}{\sum_{j=1}^{k} \frac{n_j}{\text{var}(\mathcal{D}_j)}}$. By incorporating the optimal weight into Equation (7), the sampler combination problem is reduced as follows:

$$\begin{aligned} \underset{n_1;\cdots;n_k}{arg\,min} \quad & \frac{1}{\sum_{i=1}^{k} \frac{n_i}{\text{var}(\mathcal{D}_i)}} \\ \text{subject to} \quad & \sum_{i=1}^{k} n_i = n \\ & n_i \geq 0;\ \text{for all}\ i \in [1;k] \end{aligned} \tag{8}$$

Thus, the key to solve the sampler combination problem is how to solve the budget allocation problem.

3.2 Exploration and Exploitation Trade-off

We first propose the optimal solution to the budget allocation problem and prove its optimality. However, please note that this solution cannot be achieved in practice, hence we call it *ideal* solution. We then explain the reason and find an interesting trade-off between the exploration and exploitation to develop a practical solution.

Ideal Solution. Given a query, different samplers produce estimators with different qualities. Intuitively, allocating more budget to a 'good' sampler will lead to a more accurate combined estimator. This intuition inspires us to consider an extreme case which allocates all the budget to the 'best' sampler.

Specifically, we use $\text{var}(\mathcal{D}_i)$ (see Definition 2) to measure how 'good' a sampler is. The smaller $\text{var}(\mathcal{D}_i)$ is, the better the sampler is. Let S_{i^*} denote the best sampler, i.e., $i^* = \arg\min_{i \in [1;k]} \text{var}(\mathcal{D}_i)$.

The ideal solution allocates all the budget to S_{i^*} and allocates zero budget to S_i (for $i \neq i^*$). That is, for each $i \in [1;n]$, we have:

$$n_i = \begin{cases} n; & \text{if } i = i^* \\ 0; & \text{otherwise} \end{cases}$$

Optimality. We then prove that the ideal solution is the optimal solution of the budget allocation problem (formalized in Equation (8)), as shown in Lemma 1.

Lemma 1. *Given a set of k samplers, a query q, and a total budget n, the optimal budget allocation is to allocate all the budget n to the best sampler S_{i^*}.*

Proof. Due to the space constraint, we omit all the proofs in this paper and put them in our technical report [19]. □

Exploration vs. Exploitation. Although we get the optimal solution of the budget allocation problem, this solution is impractical. The reason is that it requires the knowledge of the best sampler in advance, while computing each sampler quality $\text{var}(D_i)$ requires scanning the full data and against the purpose the sampling. To solve this problem, we develop a framework to approach the ideal solution, and it can allocate most of the budget to the best sampler without knowing which sampler is the best. More specifically, we use a sample to estimate $\text{var}(D_i)$ and define *empirical best sampler* as the sampler with the highest estimated sampler quality. In this way, the sample budget can be allocated with two different purposes: i) *Exploration:* it is allocated to estimate each sampler quality in order to find the best sampler; ii) *Exploitation:* it is allocated to the *empirical* best sampler in order to enhance the combined estimator.

There is an interesting trade-off between exploration and exploitation. When allocating more budget to explore sampler quality, it is more likely to find the true best sampler. However, there will be less budget left for the empirical best sampler to enhance the combined estimator. On the other hand, when allocating less budget to explore sampler quality, it is more likely to estimate sampler quality incorrectly and regard a bad sampler as the best sampler. As a result, most of the budget could be allocated to this bad sampler.

3.3 Model as Multi-Armed Bandit

The well-known Multi-Armed Bandit (MAB) problem also faces the exploration and exploitation trade-off [40]. One natural question is that can we borrow some ideas from MAB to solve our problem? In this subsection, we first introduce some background about MAB and then present how to model our problem as MAB. Finally, we justify why this modeling makes sense.

Background. MAB is a classical reinforcement learning problem that studies the exploration and exploitation trade-off. Consider a slot machine with k arms. A player needs to decide which arm to pull at each round. If arm i is pulled, it will return a random reward, usually in $[0; 1]$, from an unknown reward distribution $\mathcal{D}_i$ specific to arm i. The goal of the player is to maximize the total reward, or minimize the regret (will be defined later), in n rounds.

Let $\mu_i = \mathsf{E}[\mathcal{D}_i]$ and $\mu^* = max_{i \in 1:::k}\{\mu_i\}$. Clearly, the optimal strategy is to always pull the arm with the highest expected reward. However, since we do not know the best arm in advance, there exists a 'regret' of the actual pulling strategy compared to the optimal strategy. The regret over n rounds, denoted by R_n, is defined as the difference of the total rewards achieved by the optimal pulling strategy and by the actual pulling strategy, i.e.,

$$R_n = \mu^* \cdot n - \sum_{t=1}^{n} \mu_{I_t}; \tag{9}$$

where I_t is the chosen arm in round t. The expected regret is $\mathsf{E}[R_n] = \mu^* \cdot n - \sum_{t=1}^{n} \mathsf{E}[\mu_{I_t}]$. We denote $\Delta_i = \mu^* - \mu_i$ by the reward gap between an arm i and the best arm, and rewrite $\mathsf{E}[R_n]$ in terms of Δ_i:

$$\mathsf{E}[R_n] = \sum_{i=1}^{k} \Delta_i \cdot \mathsf{E}[T_i(n)] \; ; \tag{10}$$

where $T_i(n)$ is the number of times that an arm i is pulled over n rounds, and clearly, they satisfy $\sum_{i=1}^{k} T_i(n) = n$.

The goal is to find a pulling strategy to minimize $\mathsf{E}[R_n]$. For a given arm i, if it is an optimal arm, i.e., $\mu^* = \mu_i$, then we have $\Delta_i = 0$, thus no matter how many times the optimal arm is pulled, there is no impact on $\mathsf{E}[R_n]$. Thus, only the number of times that each sub-optimal arm is pulled will affect $\mathsf{E}(R_n)$. There are several pulling strategies proposed in the MAB literature [40]. They provide good theoretical guarantees. When using these strategies, the number of times that each sub-optimal arm is pulled can be upper-bounded by $O(\ln n)$.

Modeling. We next discuss how to model the sampler combination problem as MAB. Each sampler can be regarded as an arm in MAB. At each round, we need to pick up one sampler and draw a small sample from it. This can be thought of as picking up an arm and pulling it to get a random reward.

The *key difference* between the two problems is the definition of the regret (i.e., the objective function). For MAB, as shown in Equation (9), the regret is a linear combination of the observed random reward of each round. For our problem, the regret measures the difference of the estimator variances between an allocation strategy and the optimal strategy (i.e., allocating all the budget to the best sampler), which is defined as

$$R_n = \frac{1}{\sum_{i=1}^{k} \frac{n_i}{\text{var}(\mathcal{D}_i)}} - \frac{\text{var}(\mathcal{D}_{i^*})}{n}; \tag{11}$$

where i^* is the index of the best sampler, n_i is the size allocated to sampler S_i, and n is the total sample size.

Justification. Although our problem has a different objective function than the MAB problem (Equation (9) vs. Equation (11)), the two objective functions share two common properties, which make it possible to apply MAB-based approaches to solve our problem. First, their optimal solutions are the same, i.e., allocating all the budget to the best sampler (or arm), which lead to zero regret.

Property 1. *The optimal solution to the sampler combination (or MAB) problem is to allocate all the sampling (pulling) budget to the best sampler (or arm).*

In reality, however, the optimal solution of sampler combination problem cannot be achieved since the best sampler is unknown. This is similar to MAB setting where the arm quality is unknown and needs to be estimated from each pull. It inspires us to solve the problem in an iterative framework similar to MAB: explore the sampler quality and exploit budgets to the empirically best sampler.

We also interestingly observe that for both objective functions, the optimal action is independent of what actions have been taken in the previous iterations. That is, it is always optimal to choose the best sampler (or arm) at every individual iteration, regardless of which samplers were chosen in the previous iterations.

Property 2. *At each iteration, the optimal action for the sampler combination (or MAB) problem is to choose the best sampler (or arm), which is independent of historical actions.*

This property implies that MAB and SamComb follows the same principal to take action in each iteration: choose the best one in each iteration. Therefore, if we apply an MAB-like strategy, the

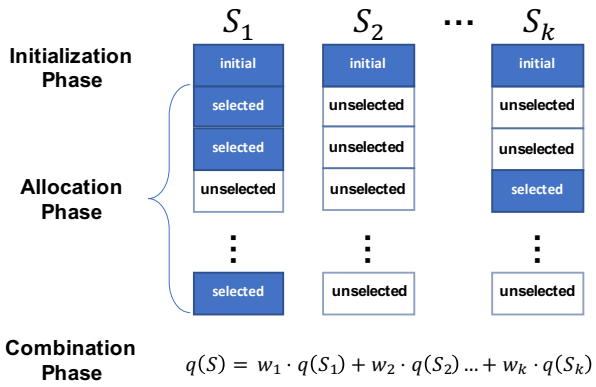


Figure 2: The SamComb Framework

action it takes may also lead us to its guarantee: most budgets are allocated to the best sampler, which can help achieve our goal.

3.4 Framework

Since our problem and MAB shares the same goal, i.e., choose the best sampler (arm) as many times as possible. Thus, it makes sense to model our problem as MAB. To this end, we propose SamComb, an MAB-based sampler combination framework.

We first introduce the existing MAB framework. Initially, the framework pulls each arm once, to get an initial estimation of the arm quality. After that, it decides which arm to pull in an iterative scheme. The fundamental challenge is how to balance the exploration and exploitation trade-off. Well-known strategies, such as ϵ-greedy and UCB [5], are proposed to handle the trade-off systematically, and guarantee that the best arm is pulled as many times as possible.

The framework of SamComb is shown in Figure 2. It is an iterative framework based on MAB, and consists of three phases:

(1) *Initialization Phase:* In the initialization phase, SamComb draws an initial batch of tuples from each sampler, to get an initial estimation of sampler quality.
(2) *Allocation Phase:* In the allocation phase, at each iteration, SamComb decides which sampler to select in order to achieve the best trade-off between exploration and exploitation and then draws a small batch of tuples from it. We extend the ϵ-greedy and UCB strategies from MAB and prove their theoretical guarantees in Section 4. The allocation phase stops once the budget is exhausted.
(3) *Combination Phase:* After the allocation phase, suppose each sampler is allocated a sample of n_i tuples such that $\sum_{i=1}^{k} n_i = n$. In the combination phase, SamComb computes an estimator $q(S_i)$ from each sampler and combines them (by computing a weighted average) into the final estimator. We propose two weight allocation approaches in Section 5.

4 ALLOCATION PHASE

We first present the ϵ_t-greedy strategy in Section 4.1 and then the Lower Confidence Bound (LCB) strategy in Section 4.2. We prove that both strategies can guarantee that the allocated sample size to any sub-optimal sampler is at most $O(\ln n)$, where n is the total sampling budget.

4.1 ϵ_t-greedy

ϵ_t-greedy [5] is a simple but powerful approach in MAB. It controls the exploration and exploitation trade-off through a parameter ϵ_t: at each iteration t, it pulls a random arm with a probability of ϵ_t (exploration), and pulls the empirical best arm with a probability of $1 - \epsilon_t$ (exploitation).

A simple approach is to set ϵ_t to a fixed value, i.e., keeping the same trade-off at every iteration. However, it contradicts the intuition that we should explore more in early iterations to look for the best arm and then exploit more in later iterations once the empirical best arm is more likely to be the true best arm.

A more sophisticated approach is proposed to address this issue [5]. It progressively decreases ϵ_t as iteration t increases. I.e., $\epsilon_t = \min\{1; \frac{ck}{d^2 t}\}$, where t refers to the t-th iteration, k is the number of arms, and c and d are user-specified parameters. It is proved that when $c > 5$, and d is upper-bounded by the reward gap between the best arm and the second best arm (i.e., $0 < d \leq \min_{i:\mu_i<\mu^*}\{\Delta_i\}$, where Δ_i is the reward gap between the best arm and arm i), ϵ_t-greedy achieves a logarithmic regret [5]. It is much better than the strategy of using a fixed ϵ.

We extend the ϵ_t-greedy strategy to solve our problem. Algorithm 1 shows the pseudo-code. The main challenges are i) how to identify the empirical best sampler, and ii) how to prove the theoretical guarantee of our ϵ_t-greedy strategy.

Empirical Best Sampler. We start with the first challenge. At each iteration, we need to identify the empirical best sampler, which requires estimating the sampler quality. In MAB, the arm quality is estimated by averaging the random reward for each pull of the arm. However, directly applying this approach does not work. This is because MAB only pulls the arm once in each iteration to get an estimation (i.e., the random reward), but the sampler quality is a variance and cannot be estimated using a single tuple. To get multiple tuples, one may consider reusing tuples from previous iterations. Unfortunately, this approach loses the independence of the estimations between different iterations, which makes Hoeffding's inequality fail and lose the guarantee provided by MAB. To solve the above issue, we propose a batch-based approach: for each iteration, we pull a batch of tuples to estimate the sampler quality, then we average the estimations of different iterations to get the final estimation. In this way, we can estimate the sampler quality in each iteration, and also preserve the independence between iterations.

Let ΔS denote a batch of tuples randomly drawn from sampler S. The estimated sampler quality using ΔS is computed as [2]:

$$\mathrm{S.batchEst} = \frac{1}{|\Delta S|-1} \cdot \sum_{i=1}^{|\Delta S|} \left(\frac{y_i}{p_i} - q(\Delta S)\right)^2 \tag{12}$$

Note that ΔS could be empty if the current budget is not allocated to the sampler. In this case, S.batchEst is set to 0.

After getting an estimation of the sampler quality from each batch, we then average them to get the final estimation. The sampler

[2] See Theorem 4.2.3 in [39]. Note that it needs to be multiplied by n to get our estimator.

Algorithm 1: ϵ_t-greedy strategy

Input : A set of samplers $\Psi = \{S_1; S_2; \cdots; S_k\}$, budget n, batch size b, parameters c and d
Output: A set of samples $\psi = \{S_1; S_2; \cdots; S_k\}$

```
# Initialization Phase
for each sampler S in Ψ do
    ΔS: Draw a batch of b tuples from S ;
    S:batchEst = 1/(|ΔS|−1) · Σ_{i=1}^{|ΔS|} (y_i/p_i − q(ΔS))^2 ;
    S.batchEstSum = S:batchEst ;
    S.batchNum = 1;
    S.sample = ΔS
end
# Allocation Phase
for t = 1 to n=b do
    ε_t = min{1; ck/(d^2 t)} ;
    if rand() > ε_t then
        S* = arg min_{S∈Ψ} S.batchEstSum / S.batchNum ;
    end
    else
        S* = a random sampler from Ψ;
    end
    ΔS: Draw a batch of b tuples from S* ;
    S*:batchEst = 1/(|ΔS|−1) · Σ_{i=1}^{|ΔS|} (y_i/p_i − q(ΔS))^2 ;
    S*.batchEstSum + = S*:batchEst ;
    S*.batchNum += 1;
    S*.sample + = ΔS ;
end
ψ = {S:sample | for each S ∈ Ψ} ;
return ψ
```

with the minimal average value is identified as the empirical best sampler at the t-th iteration, i.e.,

$$S^* = \arg\min_{S \in \Psi} \frac{\sum_{i=0}^{t} S{:}\text{batchEst}_i}{S{:}\text{batchNum}};$$

where batchNum is the total number of batches that have been allocated to this sampler after t iterations.

Theoretical Guarantee. We theoretically analyze how well our ϵ_t-greedy strategy works compared to the optimal allocation strategy, which allocates all the budget to the best sampler. In MAB, ϵ_t-greedy is proved to have a logarithmic regret bound, and a sub-optimal arm is pulled at most $O(\ln n)$, given a total pulling times of n. We find that a similar theoretical guarantee also holds in our problem. Similar to MAB, the theoretical guarantee requires the estimation from each batch bounded. We use u_i to denote the bound, which is computed as $u_i = \max_{j \in 1:::N}\{(\frac{y_j}{p_{ij}})^2\} - \min_{j \in 1:::N}\{(\frac{y_j}{p_{ij}})^2\}$. I.e.,

$$0 \le S{:}\text{batchEst} \le u_i$$

In Lemma 2, we prove that the budget allocated to a sub-optimal sampler is bounded by $O(\ln n)$, given a total budget n, when $c > 5$ and $0 < d \le \min_{i \neq i^*} \frac{\Delta_i}{u_i}$, where Δ_i is the quality gap between the best sampler and sampler i.

Lemma 2. *Given a total budget n, if ϵ_t-greedy is running with parameters $c > 5$ and $0 < d \le \min_{i \neq i^*} \frac{\Delta_i}{u_i}$, then the budget allocated to any sub-optimal sampler is at most $O(\ln n)$.*

Algorithm 2: LCB strategy

Input : A set of samplers $\Psi = \{S_1; S_2; \cdots; S_k\}$, budget n, batch size b, bound u_i
Output: A set of samples $\psi = \{S_1; S_2; \cdots; S_k\}$

```
# Initialization Phase is the same as Algorithm 1
# Allocation Phase
for t = 1 to n=b do
    S* = arg min_{S_i} S_i.batchEstSum / S_i.batchNum − u_i sqrt( ln t / S_i.batchNum )
    ΔS: Draw a batch of b tuples from S* ;
    S*:batchEst = 1/(|ΔS|−1) · Σ_{i=1}^{|ΔS|} (y_i/p_i − q(ΔS))^2 ;
    S*.batchEstSum + = S*:batchEst ;
    S*.batchNum += 1;
    S*.sample + = ΔS ;
end
ψ = {S:sample | for each S ∈ Ψ} ;
return ψ
```

4.2 LCB

We propose the Lower Confidence Bound (LCB) strategy in this section. It is inspired by the Upper Confidence Bound (UCB) [5] strategy in MAB.

UCB does not only estimate arm quality but also its confidence bound, where the confidence bound represents the uncertainty of the estimation. The key idea of UCB is to be optimistic about the uncertainty of the estimation. More specifically, it pulls the arm with the highest upper confidence bound of the estimation of arm quality, i.e., the arm with the highest quality in the optimistic case. For the selected arm, there exists two cases: 1) if it is the best arm, then this is exactly what we want; 2) if it is not the best sampler, then pulling it will increase our confidence on its sampler-quality estimation (i.e., decreasing the size of the confidence bound). As a result, it is less likely to be selected in future. Hence, UCB decreases the probability of pulling a sub-optimal arm, and exploits more and more on the best arm as iteration increases. It has been proved that UCB achieves a logarithmic bound on regret [5].

We extend UCB to our problem. Since the smaller the $\text{var}(\mathcal{D})$, the better the sampler quality, we propose a strategy named Lower Confidence Bound (LCB). Similar to the reason described in Section 4.1, we cannot directly apply UCB and we need to preserve the independence between estimations of each iteration, hence we also adopt a batch-based approach rather than drawing a single tuple in each iteration. The pseudo code is shown in Algorithm 2. LCB shares a similar idea with UCB: at each iteration, it estimates the sampler quality and computes the confidence bound of the estimation. Then, it allocates the budget to the sampler with the lowest lower confidence bound, i.e., the sampler with the highest quality in the optimistic case. LCB faces two new challenges: 1) how to compute the lower confidence bound of the estimation of sampler quality? 2) can we derive a similar theoretical guarantee like Lemma 2 for LCB?

Lower Confidence Bound. We start with the first challenge. Essentially, the confidence interval measures how far a random variable is from its expectation, which could be computed from concentration inequalities. For UCB, it regards each random reward

as a random variable and applies Hoeffding inequality, as stated in Lemma 3, to compute the confidence interval.

Lemma 3 (Hoeffding Inequality). *Let $X_1; : : : ; X_n$ be independent random variables bounded by the interval $[a_i; b_i]$: $a_i \le X_i \le b_i$. Let $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$, then Hoeffding Bound states that:*

$$Pr(|\overline{X} - E[\overline{X}]| \ge \lambda) \le 2exp(-\frac{2n^2\lambda^2}{\sum_{i=1}^{n}(b_i - a_i)^2}) \tag{13}$$

In Lemma 3, $(\overline{X} - \lambda; \overline{X} + \lambda)$ represents the confidence interval, and $2exp(-\frac{2n^2\lambda^2}{\sum_{i=1}^{n}(b_i-a_i)^2})$ refers to the confidence level. To extend UCB to LCB, we also adapt the Hoeffding inequality. Note that Hoeffding inequality requires the random variables to be bounded and independent from each other. Hence, we estimate the sampler quality for each batch, as described in Section 4.1, and regard that estimation (i.e., S:batchEst) as a random variable. Each random variable is bounded by the interval of $[0; u_i]$. We then use the average of all random variables as the final estimation of the sampler quality, and denote it by $\hat{\text{var}}(\mathcal{D})_t$, i.e.,

$$\hat{\text{var}}(\mathcal{D})_t = \frac{\sum_{i=1}^{t} \text{S:batchEst}_i}{\text{S:batchNum}}$$

Next we compute the confidence interval of $\hat{\text{var}}(\mathcal{D})_t$ by applying Hoeffding inequality. The variable number n in Hoeffding inequality (Equation (13)) is our batch number, i.e., S:batchNum. We use σ_t to represent the confidence level at iteration t, and replace λ_t with σ_t in Equation (13). Then the confidence interval is $\lambda_t = u_i\sqrt{\frac{\ln\frac{2}{\sigma_t}}{2\text{S:batchNum}_t}}$. I.e.[3],

$$Pr\left(|\hat{\text{var}}(\mathcal{D})_t - \text{var}(\mathcal{D})| \ge u_i\sqrt{\frac{\ln\frac{2}{\sigma_t}}{2\text{S:batchNum}_t}}\right) \le \sigma_t \tag{14}$$

We set σ_t to $\frac{2}{t^2}$ to make $\sum \sigma_t$ converged. Hence we compute the lower confidence bound of each sampler S at iteration t as follows:

$$\text{lcb}_t = \hat{\text{var}}(\mathcal{D})_t - u_i\sqrt{\frac{\ln t}{\text{S:batchNum}_t}} \tag{15}$$

It satisfies:

$$Pr\left(\text{lcb}_t \ge \text{var}(\mathcal{D})\right) \le \sigma_t \tag{16}$$

That is, with high probability, the lower confidence bound is no larger than the real sampler quality.

Theoretical Guarantee. We discuss how to address the second challenge, i.e., what theoretical guarantee can we get for LCB? In MAB problem, UCB strategy is proved to pull a sub-optimal sampler at most $O(\ln n)$ times, and achieve a logarithmic regret. We find that a similar bound also holds for LCB. In Lemma 4, we show that the budget allocated to a sub-optimal sampler is bounded by $O(\ln n)$.

Lemma 4. *Given a total budget n, if applying the LCB strategy, the budget allocated to any sub-optimal sampler is at most $O(\ln n)$.*

AdaptiveLCB. In LCB, we need u_i, which is the bound of the sampler-quality estimation from each batch, such that the Hoeffding inequality can be applied and the theoretical analysis holds. However, u_i could be very large. When the budget is limited, a large u_i may make LCB spend too much budget on exploration, thus decreasing the overall performance.

[3]Note that $\hat{\text{var}}(\mathcal{D})_t$ is an unbiased estimator and we have $E[\hat{\text{var}}(\mathcal{D})_t] = \text{var}(\mathcal{D})$.

Algorithm 3: AdaptiveLCB strategy

Input : Sampler set $\Psi = \{S_1; S_2; \cdots; S_k\}$, budget n, batch size b
Output: A set of samples $\psi = \{S_1; S_2; \cdots; S_k\}$
1 Insert "$u_i = S_i$:batchEst" between Line 1 and 2 in Algorithm 2
2 Insert "$u_{i^*} = max(u_{i^*}; S^*$:batchEst)" between Line 9 and Line 10 in Algorithm 2

To solve this issue, we propose a variation of LCB named AdaptiveLCB. Instead of using a large u_i that can bound all possible estimations from each batch (both historical and future estimation), we set u_i adaptively such that it can bound all the historical estimations. More specifically, we record the current maximal value of the sampler-quality estimation in each iteration, and set it to u_i. Algorithm 3 shows the pseudo code of AdaptiveLCB. We only need to add two lines of code into Algorithm 2.

Example 2 illustrates how AdaptiveLCB works.

Example 2. *Suppose we have two samplers S_1 and S_2, and the batch size is 100. We first draw a batch of 100 tuples from each sampler, and use it to estimate the sampler quality. Suppose the estimated quality for S_1 and S_2 are 10000 and 20000, respectively. Since u_i is the adaptive bound of the sampler-quality estimation, u_1 and u_2 are initialized as 10000 and 20000, respectively.*

In the first iteration, the lower confidence bound for S_1 and S_2 are $10000 - 10000\sqrt{\frac{\ln 1}{1}} = 10000$ and $20000 - 20000\sqrt{\frac{\ln 1}{1}} = 20000$, respectively (see Equation (15)). Hence, S_1 is the current empirical best sampler. We draw a batch of 100 tuples from S_1 and use it to compute a sampler-quality estimation. Suppose it is 50000. Then the average sampler-quality estimation of S_1 is updated as $\frac{10000+50000}{2} = 30000$ and u_1 is updated as $\max\{u_1; 50000\} = 50000$.

In the second iteration, the lower confidence bound for S_1 and S_2 are $30000 - 50000\sqrt{\frac{\ln 2}{2}} = 565$ and $20000 - 20000\sqrt{\frac{\ln 2}{1}} = 3349$, respectively. S_1 is still the empirical best sampler, thus we allocate a batch to S_1.

Repeat the iterative process until the budget is exhausted.

5 COMBINATION PHASE

Once the budget is exhausted, the allocation phase is finished. Now SamComb enters the combination phase. Let $\psi = \{S_1; S_2; \cdots; S_k\}$ denote the sample set derived from our allocation strategy (ϵ-greedy or LCB). The goal of the combination phase is to assign a weight to each sampler such that the variance, $\text{var}(q(\psi))$, of the combined estimator is minimized.

Section 3.1 presents the optimal solution to this weight allocation problem. However, the optimal weight requires knowing $\text{var}(D_i)$ (for each $i \in [1; k]$), which is *not* available in reality. One straightforward solution is to estimate $\text{var}(\mathcal{D}_i)$ using the allocated sample and then apply this optimal weight allocation. We call such approach *pseudo-optimal allocation*, where the weight is computed as follows:

$$w_i = \frac{n_i \hat{=} \text{var}(\mathcal{D}_i)}{\sum_{j=1}^{k} n_j \hat{=} \text{var}(\mathcal{D}_j)}; \tag{17}$$

where $\hat{\text{var}}(\mathcal{D}_i)$ is an estimation of $\text{var}(\mathcal{D}_i)$ based on S_i.

Let ψ_* denote the sample set derived from the optimal budget allocation strategy, i.e., allocating all the budget to the best sampler. Let $\mathrm{var}(q(\psi_*))$ denote the corresponding variance. We use the following formula to measure the gap of SamComb to the optimal strategy:

$$\partial ap(q(\psi)) = \frac{\mathrm{var}(q(\psi)) - \mathrm{var}(q(\psi_*))}{\mathrm{var}(q(\psi_*))} \tag{18}$$

Lemma 5 proves that SamComb is asymptotically optimal under the pseudo-optimal allocation, when $\hat{\mathrm{var}}(\mathcal{D}_i) = \mathrm{var}(\mathcal{D}_i)$ for each $i \in [1; k]$.

Lemma 5. *Under the assumption that $\hat{var}(\mathcal{D}_i) = var(\mathcal{D}_i)$ for each $i \in [1; k]$, our framework SamComb, which uses ϵ-greedy or LCB for budget allocation and uses the pseudo-optimal allocation strategy for weight allocation, is asymptotically optimal.*

We next explore an alternative weight allocation strategy to relax the assumption in Lemma 5. Obviously, a good sampler should receive a higher weight than a bad one. Thus, the key challenge is how to find an alternative way to assess sampler quality. We observe that after the budget allocation phase , the better the sampler, the larger sample size it tends to receive, since this is what our budget allocation strategy tries to optimize for. Thus, the received sample size is an indirect way to assess sampler quality. Based on this idea, we propose *proportional-to-size allocation*,

$$w_i = \frac{n_i}{\sum_{i=1}^{k} n_i}; \tag{19}$$

where n_i is the sample size received by the sampler i after the budget allocation phase.

Lemma 6 proves that SamComb is asymptotically optimal under the proportional-to-size allocation.

Lemma 6. *Our framework SamComb, which uses ϵ-greedy or LCB for budget allocation and uses the proportional-to-size allocation strategy for weight allocation, is asymptotically optimal.*

We experimentally compare the two weight allocation strategies and find that they have similar performance. Since the proportional-to-size allocation provides a nice theoretical guarantee, SamComb uses it by default.

6 EXTENSIONS

In this section, we discuss how to extend our framework to support other aggregate functions, group-by queries, and data updates.

Other Aggregate Functions. Previously we mainly discussed SUM-like query. Actually SamComb can be extended to support more aggregate functions, such as MIN, MAX and PERCENTILE query.

We first introduce how to estimate the answer using a single sampler. Suppose a tuple is sampled with a probability of 0.1, then it approximately represents 10 such tuples in the population. In this way, we can "reconstructed" the population and issue the original query over the "reconstructed" population to get an estimation. This idea is similar to the plug-in approach in [29]. Note that the "reconstructed" usually happened virtually. For many aggregation functions we do not need to actually rebuild the population. Take q-percentile (e.g., q = 0:9 represents the 90-th percentile) query as an example. As each tuple t in the sample represents $1/p_t$ tuples in the "reconstructed" population, where p_t is drawn probability of tuple t, we denote its weight as $1/p_t$. Then we sort the sampled tuples by their values, and accumulating the weights. The q-percentile query can be estimated as the largest value when the accumulated weights is no larger than q multiply by the total weights.

Now we discuss the case of multiple samplers. To be supported by SamComb, the aggregate function needs to specify: 1) how to allocate the budget in each iteration and 2) how to combine the results from multiple samplers.

For the first question, since most aggregate functions can not compute the confidence bound easily or efficiently, we adopt the ϵ-greedy strategy by default. To compare samplers' qualities, by default we compare the selectivity in their sample. One may also use different approaches for different functions. E.g., for MAX query, we can simply compare the max value in the sample (exclude the tuples which do not satisfy the predicate), since the sampler with a larger max value definitely makes a better estimation.

For the second question, since linear combination (by w_i) does not work for all the aggregate functions, a similar idea of reconstructing the population can be applied to the general case. For example, to process a median query, we can reconstruct multiple populations from different samplers, and then merge them and choose the median value of the merged reconstructed population.

Group-by Queries. To process group-by query, we will maintain the related information for each group and apply previously discussed approaches to estimate each group. Then, the left question is how to select the sampler in each iteration. To answer this question, we consider minimizing the max error among groups. For SUM-like queries, we use the estimator variance to measure the group error and select the sampler that has the highest quality for the group with the max error. For other aggregate queries (e.g., percentile), we use # of tuples to measure the group error. Let g be the group with the minimal tuples, we then select the sampler that are more likely to contain tuples in g.

Data Update. In this work we focused on insert-only update, like many existing works [34]. When data is updated, SamComb will maintain the pre-computed samples. Let D_Θ be the stale data, S_Θ be the stale sample created by a sampler $\mathcal{S}$, and D_Δ be inserted data, respectively. Now, the goal is to maintain S_Θ such that it is equivalent to a sample with the same size drawn by $\mathcal{S}$ from $D_\Theta \cup D_\Delta$. To make the maintenance efficient, the high level idea is to replace some tuples in S_Θ with a sample drawn from D_Δ. I.e., the new sample consists of two parts: the kept tuples in S_Θ, denoted as S'_Θ; and the drawn sample from D_Δ, denoted as S_Δ. For example, a uniform sampler draws each tuple with probability $p_i = \frac{1}{N}$. In sample S_Θ, each tuple is drawn with probability $\frac{1}{N_\Theta}$. After data is updated, each tuple should be drawn with probability $\frac{1}{N_\Theta+N_\Delta}$. Hence, we keep each tuple in S_Θ with probability $\frac{N_\Theta}{N_\Theta+N_\Delta}$ to get S'_Θ, and draw each tuple in D_Δ with probability $\frac{1}{N_\Theta+N_\Delta}$ to get S_Δ. Finally, the updated sample is $S'_\Theta \cup S_\Delta$.

7 EXPERIMENT

We evaluate SamComb using both synthetic and real datasets. The experiments aim to answer the following questions:

- What should be the best setting for SamComb?
- Is it necessary to combine multiple samplers?

- Is bandit-based better than heuristic-based?
- How does SamComb perform in various settings?

7.1 Experimental Setup

Datasets. 1) *TPCH-Skew* is a synthetic dataset generated from a variation of the TPC-H benchmark [9]. We set parameters skewness $z = 2$ and scale $s = 1$, and focused on the lineitem table, which contains 6 million rows and 16 columns. We also generated a larger dataset using scale $s = 10$ to test the end-to-end performance. 2) Loan [1] is a real-world peer-to-peer loan dataset. It concatenated historical loans from Prosper and Lending Club from 2013 to 2018. The dataset contains 3 million rows and 18 columns with a wide variety of data distributions.

Samplers. We created a uniform sampler, a stratified sampler on `l_returnflag`, and a measure-biased sampler on `l_extendedprice` for *TPCH-Skew* dataset, and created a uniform sampler, a stratified sampler on `grade`, and a measure-biased sampler on `principal_balance` for *Loan* dataset. Note that a few tuples may have a very large value and thus be selected too many times by the measure-biased sampler, hence we will binnizate the measure column before computing the sampling probability.

Queries. The population parameters that we aim to estimate are in the form of `SELECT SUM(A) FROM table WHERE Condition(B1) and Condition(B2), ..., Condition(Bk)`. The query selectivity is between 0:1% and 1% of the population. For each query, the aggregation column `A` and each condition column `Bi` are randomly selected, and the number of condition columns is a random number between 1 and 5. `Condition(Bi)` is in the form of $x \leq$ `Bi` $\leq y$ and `Bi` $= x$ for numerical column and categorical column, respectively. Obviously, if there is no difference between sampler qualities, then there is no need to decide which sampler to select. Thus, we generated two groups of queries based on the quality gap, named Small Gap Query and Large Gap Query, where each group has 50 queries. For Small Gap Query and Large Gap Query, the ratio of the quality of the second best sampler and the best sampler are smaller than 1.5, and larger than 1.5, respectively.

Error Metric. We used relative error to measure estimation quality. Suppose the true query result is q and the estimated result is $\hat{q}$, then the error is computed as $|\frac{\hat{q}-q}{q}|$. To reduce the randomness of estimation, we run each query over 5 different samples and compute the average error. Given a set of queries, we reported their 90th percentile average error. If the error is 2%, it means that 90% of queries have an average error (of 5 runs) smaller than 2%.

Implementation and Settings. We implemented SamComb in Java. The budget was set to 144,000 by default, which was 5% of *Loan* and 2:5% of *TPCH-Skew*, respectively. The experiments were conducted on a MacBook Pro with an Intel Core i5 2.3GHz, 16GB RAM, and 250GB SSD.

7.2 Evaluation of Our Approach

In this section, we evaluate SamComb under different settings. The goal is to find the best setting for SamComb. Note that ϵ-greedy has a parameter C but LCB does not. Thus, we first find the best parameter for ϵ-greedy, and then compare ϵ-greedy with two LCB-based approaches.

Parameter Selection for ϵ-greedy. Recall that $\epsilon_t = \min\{1; \frac{ck}{d^2 t}\}$, where c and d are user given parameters [5]. For simplicity, let $C = \frac{ck}{d^2}$. As C increases, ϵ-greedy does more and more explorations.

We vary the parameter C to see how it affects the performance. The result is shown in Figure 5. We can observe that a small C usually works well. There are two reasons. First, if the sampler quality is very different from each other, then it is easy to distinguish them using a few explorations, thus a small C is preferred. Second, if the sampler quality is close to each other, choosing any sampler will not affect the performance much, thus a small C is also good. Based on this observation, we choose $C = 10$ for ϵ-greedy by default.

Comparing ϵ-greedy, LCB, and AdaptiveLCB. We compare ϵ-greedy, LCB, and AdaptiveLCB, aiming to find the best budget allocation strategy for SamComb. For a fair comparison, we randomly set the parameter for ϵ-greedy. The comparison results on the two datasets are shown in Figure 3. We have three observations. Firstly, LCB performed much worse than ϵ-greedy and AdaptiveLCB. This is because that LCB had a large bound and needed more sample to reduce the bound. It also shows that AdaptiveLCB got a good bound to automatically balance the exploration and exploitation trade-off. Secondly, there is a larger gap between different strategies when handling Large Gap queries than Small Gap queries. The underlying reason is that when samplers have similar qualities, the difference of selecting a different sampler is not big. Thirdly, AdaptiveLCB and ϵ-greedy had a similar performance and there is no clear winner. In TPCH-Skew, ϵ-greedy performs better in the beginning then AdaptiveLCB outperforms as budget increases. In Loan, AdaptiveLCB is slightly better.

In summary, AdaptiveLCB and ϵ-greedy outperformed LCB. Since AdaptiveLCB does not need to tune the parameter, AdaptiveLCB is chosen as the default strategy.

Varying Batch Size. SamComb draws a batch of tuples from a sampler in each iteration. In this experiment, we vary the batch size from 10 to 10000, and evaluate its impact on error and latency. The result is shown in Figure 6.

From Figure 6a, we can observe that the error is relative stable when the batch size is not very big. Although a small batch may be less accurate for estimating the sampler quality in each iteration, it also leads to more iterations, which is good for allocating more budgets to the best sampler. As a result, the error is relative stable.

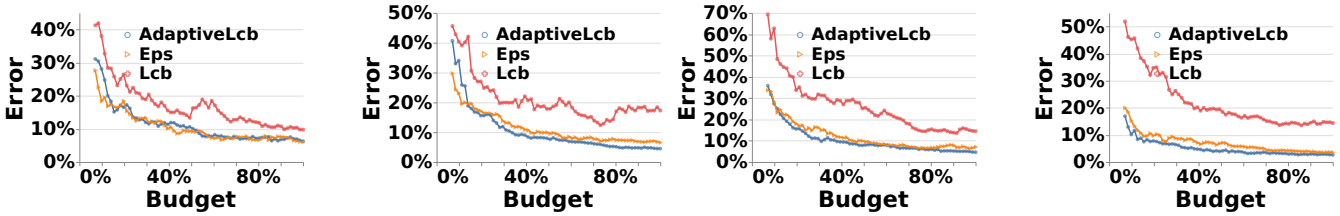
Figure 6b shows that the latency could be very high for a small batch size, but it will keep stable when the batch size is above a threshold. This is because SamComb issues a query for each batch, and a small batch size will lead to many batches, causing a big overhead. When the number of bathes is small, the overhead is negligible comparing to the query processing time, thus the latency becomes stable.

From this experiment, we can conclude that a poor setting of batch size could affects the latency a lot, while the impact on error happened slowly. Hence, we could choose the smallest batch size when increasing batch size does not further decrease the latency.

7.3 Comparison of Combination Approaches

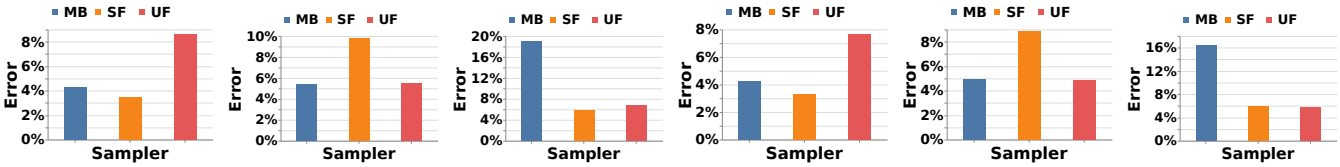
In this section, we first justify the need for sampler combination and then compare different combination approaches.

One Size Does Not Fit All. We test the performance of the 100 queries with each individual sampler, and split the queries into



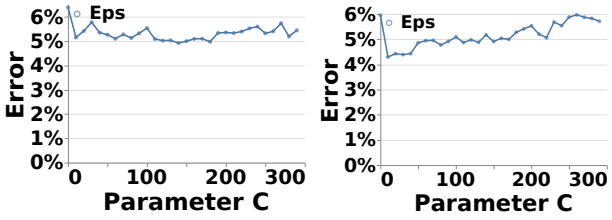
(a) TPCH-Skew, Small Gap (b) TPCH-Skew, Large Gap (c) Loan, Small Gap (d) Loan, Large Gap

Figure 3: ϵ-greedy vs LCB vs AdaptiveLCB (TPCH-Skew)



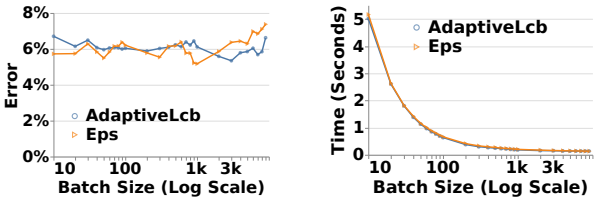
(a) TPCH-Skew, U (b) TPCH-Skew, S (c) TPCH-Skew, M (d) Loan, U (e) Loan, S (f) Loan, M

Figure 4: Justification for Sampler Combination



(a) TPCH-Skew (b) Loan

Figure 5: Parameter Selection for ϵ-greedy



(a) Error (b) Latency

Figure 6: Varying Batch Size

three groups, named *U*, *S* and *M* group. In the *U*, *S*, and *M* group, the uniform sampler, the stratified sampler, and the measure-biased sampler performed worse than the other two samplers, respectively. We plot the performance for different samplers over the three groups of queries, as shown in Figure 4.

We can see that there is no single sampler that performs well in all cases. For example, on the Loan dataset, the stratified sampler performs much better than the other two samplers in the *U* group. However, it is much worse than the others in the *S* group. A similar observation can also be derived from the TPCH-Skew dataset. These results validate that one size does not fit all for sample-based estimation, and there is a strong need to combine multiple samplers.

Comparing with Best Sampler & Worst Sampler. For each query, there is a best sampler and a worst sampler, which is unknown unless scanning the full data. In this experiment, we comparing SamComb with two approaches: i) BestAlways: always choosing the best sampler. ii) WorstAlways: always choosing the worst sampler. The purpose of this experiment is to understand how far the estimation of SamComb is close to BestAlways and WorstAlways. The result is shown in Figure 8d.

Figure 8d shows that the error of SamComb is close to BestAlways and is much more accurate than WorstAlways. This is because SamComb combines multiple samplers and allocates more budgets to the best sampler. We further justify this point by investigating the allocated tuples of SamComb to the best sampler and the worst sampler for each query. It turns out that there are 73% queries where the best sampler is allocated more than 90% of the budgets, while 95% queries where the worst sampler is allocated less than 10% budget. This result further proves that SamComb successfully allocates most budgets to the best sampler.

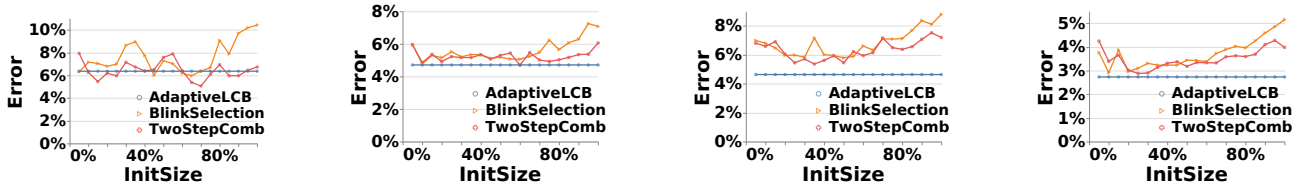
Bandit-based vs Heuristic-based. We compare our bandit-based approach with two heuristic approaches.

TwoStepComb allocates the budget in two step: it first allocates an initial budget to each sampler and estimates their qualities, and then it allocates all the remaining budget to the empirical best sampler, which is similar to the explore-first approach [40]. Finally, it combines the estimation from multiple samples.

BlinkSelection is the sample selection technique used in BlinkDB [4]. It builds the error-latency profile for each sample and chooses the one satisfying the error or latency threshold. In our scenario, the latency is proportional to sample size (since we sequentially scan the data) and the relationship of error and sample size is clear. That is, we only need to estimate the sampler quality, then the error-budget profile can be built. Hence, BlinkSelection is actually the same as TwoStepComb without the sample combination phase.

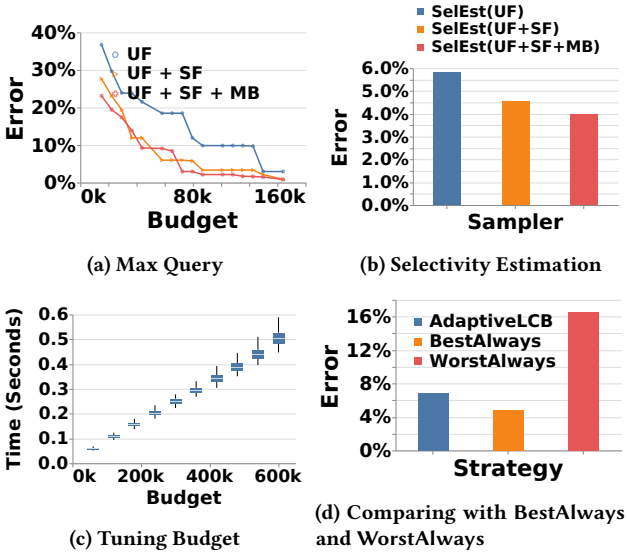
The performance of TwoStepComb and BlinkSelection depends on the initial budget size. We varied this parameter in TwoStepComb and BlinkSelection, and compared them with SamComb. Figure 7 shows the result. We see that SamComb outperformed TwoStepComb and BlinkSelection. This is because that SamComb allocated the budget adaptively, while TwoStepComb and BlinkSelection only used the initial estimation to decide the allocation of the remaining budget.

One major issue of TwoStepComb and BlinkSelection is that their performance is sensitive to the initial size. As shown in Figure 7, the



(a) TPCH-Skew, Small Gap (b) TPCH-Skew, Large Gap (c) Loan, Small Gap (d) Loan, Large Gap

Figure 7: Comparing SamComb, TwoStepComb and BlinkSelection



(a) Max Query (b) Selectivity Estimation

(c) Tuning Budget (d) Comparing with BestAlways and WorstAlways

Figure 8: Evaluation in various settings (TPCH-Skew)

performance of TwoStepComb and BlinkSelection first increased then decreased. This is because that at the beginning, TwoStepComb and BlinkSelection did not estimate the sampler quality accurately and chose a bad sampler as the empirical best sampler. With a larger initial budget, the quality estimation became more accurate and the probability of choosing the best sampler became larger. Hence, the performance was improved. However, as the initial size became larger and larger, the remaining budget became smaller and smaller. As a result, the performance of TwoStepComb and BlinkSelection decreased and it is more like equal allocation.

The result also indicates that the initial size is hard to tune. A good setting of initial size varies from query to query, and data to data. For example, a good initial size is around 40% and 20% of the total budget in Figure 7c and Figure 7d, respectively.

7.4 Evaluation in Various Settings

Support Other Aggregation Functions. We evaluate the performance of SamComb in supporting other aggregate functions. We pick up MAX because it is a challenging one for sample-based estimation. We used the same query workload but replaced the aggregate function with MAX. We calculated the rank of the estimated max value and get its relative rank error based on $rank\_error = 1 - \frac{rank(estimate)}{N}$, where N is the population size. The result is shown in Figure 8a. We can see that SamComb can successfully combine multiple samplers and return a much more accurate answer to MAX queries compared to using a uniform sampler only. This is a promising result since it shows that combining multiple samplers enables sample-based estimation to handle difficult aggregation functions more accurately. We defer an extensive study of this direction to future work.

Table 1: Evaluation of maintenance cost (10% new data)

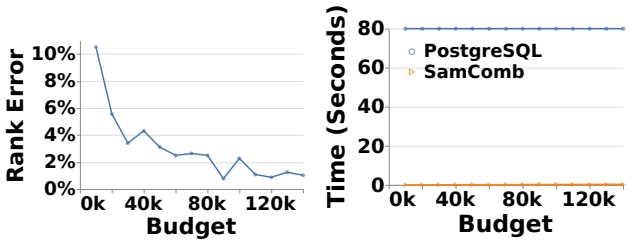
	Sample Creation	Sample Maintenance
Uniform	29.33 secs	2.64 secs
Measure-biased	29.81 secs	2.92 secs
Stratified	29.09 secs	2.63 secs

Selectivity Estimation. Selectivity estimation aims to estimate the percentage of the tuples that satisfy a predicate. It is an essential step in query optimization. Sampling is a common approach for solving this problem in existing database systems [37].

We evaluate selectivity estimation when multiple samplers are available. Selectivity estimation can be expressed using COUNT queries. Thus we replace the aggregation function in our queries from SUM to COUNT. The result is shown in Figure 8b. We can see that adding more samplers improves estimation accuracy. For example, adding the stratified sampler reduces the estimation error of the uniform sampler by 22%. We also notice that adding the measure-biased sampler improve less. This is because the measure-biased sampler is designed to sample more from the tuples with large measure values. However, in the selectivity estimation scenario, the queries are COUNT rather than SUM. Our SamComb framework automatically figured this out without wrongly selecting the measure-biased sampler to hurt the performance.

Tuning Budget. The budget size is a parameter used by AQP system to make the trade-off between the latency and error. Sometimes users may have a error or latency requirement, and want to tune the budget. This can be achieved by modeling the relationship of budget-error or budget-latency (e.g., the Error Latency Profiles in BlinkDB). We varied the budget and measured the latency of each query. Figure 8c shows the latency distribution of all queries under different budgets. We can find a linear relationship between budget and latency among different queries. This is because the query time is dominated by IO and the sample is scanned sequentially, thus the time is (approximately) proportional to the number of scanned tuples. It demonstrates that we can build a budget-latency profile to tune the budget based on the latency requirement.

Data Update. We tested the overhead of SamComb for data update (insert). We used the TPCH-Skew 1G as the original data, and took its 10% sample as the data to be inserted. The sample size is 1% of the original data for each sampler. We tested the time of creating an initial sample and the time of incrementally maintaining it. The result is shown in Table 1. We can see that the maintenance cost is relatively small. For example, the cost of maintaining a stratified sample is only $\frac{2:63}{29:09} = 9\%$ of the total sample creation time. This is



(a) Relative Rank Error (b) Time (Seconds)

Figure 9: Performance of 90-Percentile Queries

Table 2: End-to-end performance comparison (Budget = 1%)

	Query Error	Response Time
VerdictDB	2.67%	0.36 secs
SamComb	1.89%	0.44 secs
PostgreSQL	0	32.93 secs

because that incremental maintenance only needs to scan the stale sample and the delta table rather than the whole data.

7.5 End-to-end Performance

In this section, we conducted experiments on a TPCH-Skew data with scale 10 to show the end-to-end performance of SamComb.

Compare with VerdictDB and PostgreSQL. We compare the performance of SamComb with VerdictDB and PostgreSQL. The budget was set as 1% of the full data for SamComb and VerdictDB. The PostgreSQL is the approach that directly issue the query in PostgreSQL over the full data. The SamComb is built on top of PostgreSQL and each sampler is stored as a table in PostgreSQL. In each iteration, a query with predicate `row_id BETWEEN a AND b` is issued to get the statistics in the batch whose tuple id is between a and b. Then, the statistics are used to compute the estimated sampler quality.

The result is shown in Table 2. From Table 2, we can see that under the same budget, the latency of SamComb is close to VerdictDB (but slightly slower). This is because they both scan samples sequentially at query time. It also shows that the overhead of SamComb is small. The reason is that the statistics used by SamComb to select a sampler can be computed incrementally. Furthermore, comparing to regular execution in PostgreSQL, SamComb can be round 80 times faster, since it only scans a small sample rather than the full data.

Percentile Query. In this experiment, we test the performance of SamComb for answering percentile queries. We choose 90th percentile, since the median percentile can handle by uniform sample well and extreme percentile is more challenging for AQP system.

We vary the budget from 10k to 120k, and evaluate the relative rank error and time of SamComb. The result is shown in Figure 9. Figure 9a shows that the estimation quality of SamComb for extreme percentile is improved as more budgets are allocated. This is because SamComb combines multiple samplers, and some samplers could draw tuples that are important to the extreme percentile with a higher probability. From Figure 9b, we can observe that the latency of SamComb scaled approximately linearly when the budget is not very big. This is because the main cost came from the IO scan of samples, rather than sorting the elements. Comparing to PostgreSQL whose latency is 80 seconds, SamComb can be 80 times faster within an error 2%.

8 RELATED WORK

We review the related work on sample-based estimation, which can be divided into single-sample based and multiple-sample based.

Single Sample. Sample-based estimation has been extensively studied in both statistics [39] and databases [25, 30]. To estimate a population parameter, a simple approach is to draw a uniform sample and then use it to estimate the parameter. To improve the performance, various sampling techniques have been proposed. E.g., Sample+Seek [13] applies measure-biased sampling to improve uniform sampling. START [8] constructs an optimal stratified sample based on a given query workload. Congressional sampling [2] constructs a biased sample optimized for a set of group-by queries. Correlated sampling [42] constructs correlated samples based on the join key column. The main idea of these techniques is to increase the sampling probability for the tuples that are important to the result. There is another work also leverages MAB for sampling [10]. Different from us, it focuses on efficiently drawing a sample from a discrete random variable with a high degree of dependency.

Multiple Samples. There are some efforts that leverage multiple samples to improve estimation accuracy. E.g., BlinkDB [4] precomputes multiple stratified samples and selects the best one using error-latency profile. Small group sampling [6] constructs multiple small group samples and selects samples based on group-by columns. VerdictDB [34] constructs different types of samples, and selects a sample using a heuristic cost model. Quickr [20, 21] applies heuristic rules to select various samplers in the sample plan and creates samples on the fly. SPEAr [22] targets at approximate stream processing scenario and selects uniform/stratified sample using heuristic rules. Bao [27] applies MAB to pick the appropriate hint for cardinality estimation. There is other related work [41] that combines samples from different distributions for Monte Carlo rendering. To the best of our knowledge, we are the first to study how to *dynamically* select and combine samplers. We show that this novel problem can be modeled as a MAB problem, and our solution balances the trade-off between exploration and exploitation in a principal way with theoretical guarantee.

9 CONCLUSION

In this paper, we proposed a novel bandit-based framework, named SamComb, which combines multiple samplers to improve the quality of sample-based estimation. We formally defined the sampler combination problem and justified why it can be modeled as a multi-armed bandit problem. Our framework consists of three phases: (i) initialization phase, (ii) allocation phase, and (iii) combination phase. For the allocation phase, we proposed two strategies based on the well-known approaches in MAB, i.e., LCB and ϵ-greedy. We proved that they both allocates at most $O(\ln n)$ budget to each sub-optimal sampler. For the combination phase, we proposed two weight allocation strategies to combine estimators, and proved that SamComb under the proportional-to-size allocation is asymptotically optimal. We extensively evaluated our approaches on both synthetic and real world datasets. The results showed that i) SamComb using the bandit-based approach (AdaptiveLCB) achieved higher performance than heuristic-based approaches (Random, TwoStepComb and BlinkSelection); ii) SamComb helped the reduce estimation error at the same sample budget (query latency).

REFERENCES

[1] Loan dataset. https://www.kaggle.com/skihikingkevin/online-p2p-lending.

[2] 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.

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 481–492, 2014.

[4] 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 *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 29–42, 2013.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[6] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 539–550, 2003.

[7] D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[8] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9, 2007.

[9] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew. ftp.research.microsoft.com/users/viveknar/tpcdskew.

[10] Y. Chen and Z. Ghahramani. Scalable discrete sampling as a multi-armed bandit problem. In *International Conference on Machine Learning*, pages 2492–2501. PMLR, 2016.

[11] Y. Chen and K. Yi. Two-level sampling for join size estimation. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 759–774. ACM, 2017.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1115–1118. ACM, 2010.

[13] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 679–694, 2016.

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

[15] S. Han, H. Wang, J. Wan, and J. Li. An iterative scheme for leverage-based approximate aggregation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 494–505, 2019.

[16] M. H. Hansen and W. N. Hurwitz. On the theory of sampling from finite populations. *The Annals of Mathematical Statistics*, 14(4):333–362, 1943.

[17] 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.

[18] D. Huang, D. Y. Yoon, S. Pettie, and B. Mozafari. Join on samples: A theoretical guide for practitioners. *Proc. VLDB Endow.*, 13(4):547–560, 2019.

[19] J. Peng, B. Ding, J. Wang, K. Zeng, and J. Zhou. One Size Does Not Fit All: A Bandit-Based Sampler Combination Framework with Theoretical Guarantees (Technical Report), 2022.

[20] S. Kandula, K. Lee, S. Chaudhuri, and M. Friedman. Experiences with approximating queries in microsoft's production big-data clusters. *Proc. VLDB Endow.*, 12(12):2131–2142, 2019.

[21] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 631–646. ACM, 2016.

[22] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. Spear: Expediting stream processing with accuracy guarantees. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1105–1116. IEEE, 2020.

[23] T. Kraska. Northstar: An interactive data science system. *PVLDB*, 11(12):2150–2164, 2018.

[24] P. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. 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 175–186. ACM, 2007.

[25] K. Li and G. Li. Approximate query processing: what is new and where to go? *Data Science and Engineering*, 3(4):379–397, 2018.

[26] A. Mahajan and D. Teneketzis. Multi-armed bandit problems. In *Foundations and Applications of Sensor Management*, pages 121–151. Springer, 2008.

[27] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1275–1288. ACM, 2021.

[28] B. Mozafari. Approximate query engines: Commercial challenges and research opportunities. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 521–524, 2017.

[29] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 38(3):3–29, 2015.

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

[31] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki. Taster: Self-tuning, elastic and online approximate query processing. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 482–493, 2019.

[32] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow.*, 4(11):1135–1145, 2011.

[33] Y. Park, M. J. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 755–766. IEEE Computer Society, 2016.

[34] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1461–1476, 2018.

[35] Y. Park, A. S. Tajik, M. J. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 587–602, 2017.

[36] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1477–1492, 2018.

[37] C. Proteau. Guide to performance and tuning: Query performance and sampled selectivity, 2004.

[38] R. Singh and N. S. Mangat. *Elements of survey sampling*, volume 15. Springer Science & Business Media, 2013.

[39] S. Singh. *Advanced Sampling Theory With Applications: How Michael"" Selected"" Amy*, volume 2. Springer Science & Business Media, 2003.

[40] A. Slivkins et al. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2):1–286, 2019.

[41] E. Veach and L. J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In S. G. Mair and R. Cook, editors, *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1995, Los Angeles, CA, USA, August 6-11, 1995*, pages 419–428. ACM, 1995.

[42] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. Join size estimation subject to filter conditions. *Proc. VLDB Endow.*, 8(12):1530–1541, 2015.

[43] J. N. Yan, Z. Gu, and J. M. Rzeszotarski. Tessera: Discretizing data analysis workflows on a task level. In Y. Kitamura, A. Quigley, K. Isbister, T. Igarashi, P. Bjørn, and S. M. Drucker, editors, *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, pages 20:1–20:15. ACM, 2021.

[44] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 277–288, 2014.

[45] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1525–1539, 2018.