LAQy: Efficient and Reusable Query Approximations via Lazy Sampling

Viktor Sanca
EPFL
viktor.sanca@epfl.ch

Periklis Chrysogelos*
Oracle
periklis.chrysogelos@oracle.com

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

Abstract

Modern analytical engines rely on Approximate Query Processing (AQP) to provide faster response times than the hardware allows for exact query answering. However, existing AQP methods impose steep performance penalties as workload unpredictability increases. Specifically, offline AQP relies on predictable workloads to create samples that match the queries in a priori to query execution, providing reductions in query response times when queries match the expected workload. As soon as workload predictability diminishes, existing online AQP methods create query-specific samples with little reuse across queries and produce significantly smaller gains in response times. As a result, existing approaches cannot fully exploit the benefits of sampling under increased unpredictability.

We analyze sample creation and propose LAQy, a framework for building, expanding, and merging samples to adapt to the changes in workload predicates. We show the main parameters that affect the sample creation time and propose lazy sampling to overcome the unpredictability issues that cause fast-but-specialized samples to be query-specific. We evaluate LAQy by implementing it in an in-memory code-generation-based scale-up analytical engine to show the adaptivity and practicality of our framework in a modern system. LAQy speeds up online sampling processing as a function of sample reuse ranging from practically zero to full online sampling time, and from 2.5x to 19.3x in a simulated exploratory workload.

ACM Reference Format:
Viktor Sanca, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. LAQy: Efficient and Reusable Query Approximations via Lazy Sampling. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Data exploration is crucial to deriving insights and informed decisions in today's data-driven world. Visualization tools and interactive dashboards provide a convenient and rich exploration interface. Nevertheless, humans require fast responses to maintain their focus during mental tasks: Miller [26] reports a 0.1 seconds response window for users to feel the UI follows their actions and 15 seconds as a hard limit to avoid demoralization and breaking the line of thought. However, with current memory technologies providing a few hundred GBps of memory bandwidth, analytical engines cannot even process simple queries that touch more than a few GBs of data in the 0.1 seconds window, despite running on TB-sized memories.

Analytical engines have long relied on approximate query processing to reduce the data processing time [2, 4, 8, 9, 11, 19, 23, 28, 33, 36]. Offline approximate query processing methods prebuild samples to reduce the data access and processing time during query execution [2]. However, the significant savings of such approaches come at the expense of requiring predictable workloads. Such predictable workloads appear in data warehousing operations, but they mismatch interactive workloads like data exploration, where queries constantly change [22]. Online query processing generalizes offline sampling into interactive use cases. Instead of relying on query templates, it does the sampling during query execution before expensive operations. Such approaches reduce the processing time; however, sampling during query execution is a heavy operation [4, 36]. To minimize this cost, existing approaches rely on pushing lightweight, selective operations below sampling [19] and sample caching [28]. However, specializing the online sample to the current query reduces its potential to be suitable for another query. As a result, existing approaches introduce a steep trade-off between sample reuse and sampling overhead for interactive exploration.

This work introduces LAQy, a sampling-based AQP framework that increases the sample reuse opportunities while maintaining the sample creation to the same or lower cost than online sample creation. LAQy relaxes the sample matching requirements by allowing samples to match a query partially. This relaxation allows using samples that would otherwise be considered inappropriate for an incoming query – resulting in significant execution time savings. LAQy corrects the query-sample mismatch using *delta* samples: samples that LAQy uses to augment the partially matching sample with the missing pieces needed to satisfy the query approximation requirements. As a result, LAQy extends i) the effectiveness of online sampling techniques to a greater range of query workloads, ii) provides a system-level acceleration technique that maintains the theoretical sample properties, and iii) reduces the query predictability requirements that existing systems need to overcome the hardware limitations through query approximation techniques.

In summary, LAQy makes the following **contributions**:

- We identify that the current rigidness of sample matching rules highly impacts a class of workloads (Section 2). We pinpoint the resulting wasteful sample creations (Section 4) and augment the workload predictability classification [2] with a new class that often appears in exploratory workloads.

*Work done while the author was at EPFL.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

- We propose a relaxed sample-to-query matching framework (Section 3) that increases sample reuse. We show how partial matching, combined with lazy delta sample creation, i) extends the reusability of online samples and ii) reduces the online sample creation cost (Section 5).
- We outline the integration of LAQy in a state-of-the-art parallel analytical engine (Section 6). We show how through partial sample matching and lazy delta samples, LAQy accelerates online approximate query processing by up to 19.3x without loss of approximation guarantees (Section 7).

Overall, LAQy improves the efficiency of online approximate query processing systems by increasing sample reuse and bridging the gap between predictable and unpredictable predicates in approximate query processing. As a result, LAQy enables fast responses despite the query unpredictability that characterizes data exploration workloads, which previously hindered the effectiveness of query approximation methods. LAQy reduces the sample creation overhead. While most of the prior related systems use sampling in disk-based, single-threaded, and distributed setups, we investigate and address the bottleneck shift in scale-up single-machine setups with minimal engine modifications. LAQy proposes a lazy sampling algorithm to avoid costly sample creation through an in-memory-friendly architecture and judicious workload-driven sample construction and merging. This allows for speeding up the sampling operation for base relations and also enables reuse opportunities when samplers must be placed after joins, for example, when meaningful filtering and sampling dimensions are only available after joining the fact tables with dimension tables.

2 Background and Motivation

Exploratory data analysis creates hard-to-predict query patterns, yet its interactive nature requires fast response times to keep the data analyst focused and productive. Further, keeping the user engaged requires response times exceeding the underlying hardware's capabilities. As a result, analytical engines have relied on approximate query processing to reduce the data processing cost.

Online and offline AQP. To improve response times, past work [2, 4, 8, 9, 11, 19, 23, 28, 33, 36] has used offline and online approximate query processing. Offline AQP prebuilds a set of samples to reply to queries during runtime quickly. In contrast to offline AQP, which relies on predictable workloads, online AQP relies on sampling during the actual runtime. This reduces the predictability requirements for online AQP and the observed benefits, as online AQP only improves operations after the sample (e.g., an online sample is created before an expensive join or shuffling operation).

A core requirement of AQP methods is that when the query implies some grouping, all groups are represented in the output. To achieve that, variants of approximation methods often rely on stratified sampling: the systems analyze the query clauses and extract the columns that, if they are not included in the stratification key, the query output could sample out tuples. These columns are called the *Query Column Set* (QCS), and when aligned with the query requirements, they allow for strict bounds on the error of the computed aggregations [2]. Furthermore, Quickr [19] provides optimization rules for injecting the samplers in the query plan and transforming samples and their QCS requirements while pushing them across various relational operators. Conversely, the remaining non-QCS columns are called *Query Value Set* (QVS).

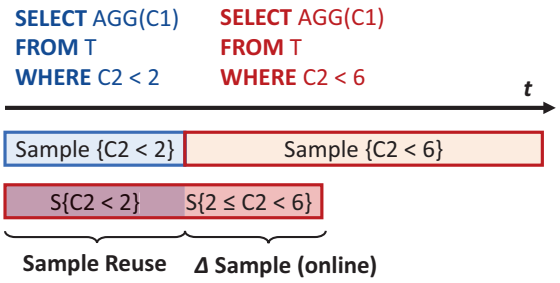


Figure 1: Relaxing the predicate predictability example.

Workload predictability. The different approximate query processing methods' applicability depends on workload predictability. Agarwal et al. [2] classify workloads into four categories based on the predictability of the queries: i) *predictable queries* where the upcoming queries are known, e.g., monthly or weekly warehousing tasks that repeat the same query in a fixed interval, ii) *predictable query predicates* where the *filter conditions* of upcoming queries are fixed and known, iii) *predictable QCSs* where the *grouping* or *filtering* columns are known, but the actual filtering values are revealed only during the query invocation, and iv) *unpredictable queries* where there is no information about the upcoming queries.

While the query patterns during data exploration are hard to predict, they are also not entirely unpredictable. Users often add, remove, expand or shrink filters, grouping columns and joined tables as a result of focusing their exploration on specific subsets of the input or expanding to increase their exploration scope and hypothesis testing [37]. As a result, the analytical engine often receives incremental changes to the query shape, placing such workloads between predictable, or slowly changing, QCSs and predictable query predicates. Yet, approximation techniques for predictable query predicates are much more efficient as samples are handled similarly to offline views and thus incur small overheads. In contrast (mostly) predictable QCSs rely on online sampling.

Issue #1: strict sample matching rules. Whether a sample subsumes a query or not is, currently, a binary classification: either a query has an appropriate QCS and QVS, or it can not provide the necessary guarantee, even if just one column is not contained in one of those sets. As a result, small mismatches in column set requirements result in creating a new sample – with the corresponding overhead and missed opportunities.

Example. Figure 1 shows two queries arriving in a sequence, with the second one having an expanded filter, similarly to the queries submitted by a data scientist who zooms out to cover a greater range of the input. Suppose that, during query execution, a sample on T is created after the filtering condition, e.g., to minimize the sampling cost. In that case, that sample will not have the information necessary to answer the expanded (red) query. As a result, while the two queries are very similar, the sample would be rejected, and a new one would be created, reducing the first sample's effectiveness.

Challenge #1: increase sample usefulness. Efficient query approximation techniques need to increase the usefulness of each

sample by relaxing the sample matching requirements, despite the theoretical requirements. To avoid compromising the theoretical sample properties, LAQy increases the sample usefulness by allowing samples to partially satisfy a query and creating delta queries that require smaller samples for their approximation.

Issue #2: creating a new sample is costly. When existing samples do not match the current query, either a new sample is created, or approximate processing is abandoned altogether. To mitigate such overheads, existing methods [28] aggressively cache samples and synopsis. Specifically, Taster [28] continuously inspects the workload to materialize samples as a side-effect of execution for future (re)use based on the recent query history. However, the materialized sample is only helpful if it entirely subsumes the predicates and QCS, again falling back to the online AQP processing otherwise.

Further, the new sample is built from scratch despite potentially having similar samples. As a result, despite an existing sample satisfying part of the query, the new sample will be built on the entire input and have the full QCS size, which prior work has shown that it can have a prohibitive cost, especially as the QCS and QVS sizes increase [36].

Example. If the queries of Figure 1 were also grouping on C2, then the corresponding samples would have C2 in the QCS. Although the blue sample is a subset of the sample required for the red query, existing approaches would discard the blue sample when evaluating the second (red) query. Further, with the valid range of C2 increasing, building the second (red) sample with stratification on C2 would be significantly slower than making the blue sample, as the number of strata is the dominating factor during sample creation [36].

Challenge #2: efficiently react to missing samples. Efficient query approximation techniques need to quickly and efficiently build the required samples. To reduce the sample creation time, LAQy judiciously builds only the missing parts of a required sample and retrieves the rest from the already available samples.

Issue #3: unpredictability increases the risk of useless pre-sampling. Offline sampling [2] mitigates the sampling overhead by sampling before the query time. During query processing, the sample is already available, effectively eradicating the cost of sampling from the response time. Further, to avoid the overhead of periodically having to recreate the sample as the source data are updated, Birler et al. [4] propose incrementally maintaining the sample during updates. Yet, offline methods require that the useful samples are known prior to query time.

To reduce the probability of wasted samples, offline approaches tend to build more inclusive and, thus, bigger samples. However, this is still a double edge sword: while it may support more queries, 1) using a bigger sample incurs a higher cost for each query using it, 2) a higher build cost and yet the potential of mispredicting the query workload and not using the sample.

Example. Consider this time that an aggressive sampling approach sees the condition on C2, and instead of applying the filter before the sampling, it decides to stratify on C2 as well (include C2 in the QCS). While this would make the sample reusable in the second (red) query, it may not pay off if, for example, multiplying the number of strata by the cardinality of C2 makes sample creation much slower than creating two different samples – a common occurrence for high cardinality columns [36].

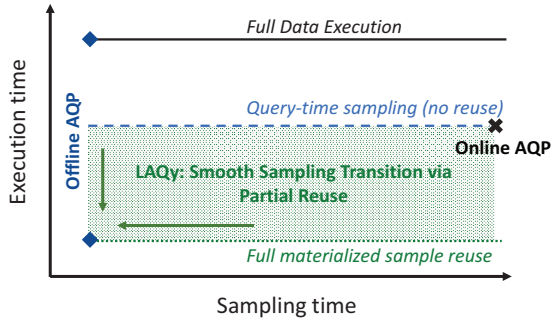


Figure 2: The design space of sampling-based AQP methods.

Challenge #3: sampling without regret. As a result, the overgeneralization of samples may result in significant costs. Efficient query approximation systems need to minimize the risk of wasteful sample creation by reducing the probability of creating a minimally useful sample. To overcome this issue, LAQy builds only samples that will be immediately used to accelerate a query.

Summary. Overall, the current rigidness of whether a sample satisfies or not a query, combined with the high cost of sample creation from scratch and the uncertainty about the long-term gains of creating bigger samples highly affect the effectiveness of online approximation techniques. Further, data exploration often results in partially overlapping queries and small transitions across predicates, resulting in the aforementioned issues being exemplified for such workloads. In the next sections, we describe how LAQy's design allows tackling all three issues through a design that focuses on maximizing the (even partial) sample reuse to minimize the sample creation (through lazy delta samples) and a sample-as-you-query model that reduces the regret for building a sample by creating only samples that are immediately useful. As a result, LAQy bridges the gap between online and offline approximation methods by partial sample reuse (Figure 2).

3 LAQy's Design Principles

LAQy is an approximate query processing engine designed to bridge the gap between offline and online AQP methods in scale-up systems. Existing approaches have to select between building low-overhead samples specialized to the query predicates or paying a higher sample construction overhead to create reusable samples. Instead, LAQy achieves a sweet spot between reusability and sampling time by taking advantage of the mergeable nature of samples. Furthermore, LAQy is compatible with adaptive storage and sample budgeting solutions, like Taster [28], to allow adaptive management of the allocated sample space based on workload patterns.

Instead of imposing a binary can-or-cannot-be reused per-sample decision, the samples generated by LAQy create a continuous spectrum between online and offline sampling methods (Figure 2). As a result, samples do not have to be entirely ignored; instead, they provide a partial input to the query. Furthermore, for the query input that is not covered by the existing sample, LAQy avoids building a

full sample. Instead, it constructs only the part of the sample necessary for the current query – minimizing the overhead imposed by sample construction, resulting in better applicability of our method in the era of in-memory engines that saturate the available memory bandwidth.

The three core principles of LAQy enable it to directly address the challenges outlined in Section 2:

Principle #1: Partial reuse. Predicates are traditionally considered either known or too volatile to specialize the sample to the predicate value – creating a steep penalty, even if the predicates are not entirely random. For example, typical data exploration patterns [37] imply that the focus of interest may change but correlate to the initial scope of analysis, albeit in unpredictable ways. LAQy exploits the overlap across the predicates to reuse the existing materialized samples and compute only the delta samples to satisfy uncovered, non-overlapping sample ranges, extending the strategy of offline sampling systems.

Principle #2: Judicious sampling. Stratified sampling is an expensive operation. Stratification imposes random accesses making sampling potentially as expensive as joins or aggregations. Nonetheless, it reduces the input to upcoming operations and the execution time of future queries if the sample is materialized and reused. To avoid excessive overheads without a corresponding long-term speedup, LAQy minimizes the input of stratification operations to the minimum required for the current query, along the direction of the online sampling systems.

Principle #3: Laziness and minimal waste. Every bit counts, but some bits count more. The long-term gain of a sample depends on upcoming queries. While the prediction accuracy of future queries can vary, samples created for the current query have an immediate turnaround. Also, we can give a higher turnaround to already created samples by increasing their utility through partial reuse. LAQy reduces the decision-making and pushes it as late as possible by building only the necessary delta samples and merging them only when needed. This makes our approach complementary to both the online and offline state-of-the-art sampling-based AQP systems.

4 Pruning the Sample Construction Space

LAQy bridges the gap between offline and online sampling by exploiting the overlap of query predicates to reuse and merge samples. Section 4.1 starts by evaluating the different parameters that affect the build time for a stratified sample. Section 4.2 links the various parameters with the query predictability. Lastly, in Section 4.3, it motivates our main observation: relaxing the predictability requirements for predicates allows lazily building samples and amortizes the (stratified) sample build time across queries with overlapping predicates.

4.1 Building Stratified Samples

To create a stratified sample, each input element is inserted into a reservoir based on the element values on the QCS columns. Each stratum thus keeps track of the reservoir, and the number of considered elements (*weight*) as the admission probability depends on the number of previously considered items.

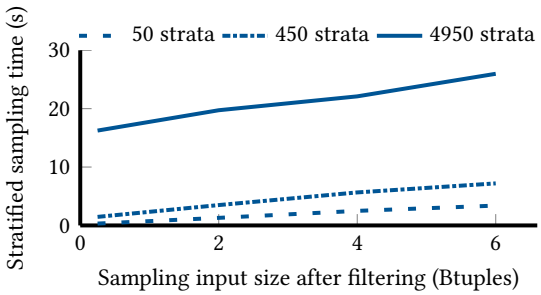


Figure 3: Impact of #tuples of the dataset and #strata defined by the QCS on the time to create a stratified sample.

Considering an input item for inclusion into a reservoir generally requires random access to find and update the *admission-control* state (*random number generator*). If the item is admitted, then finding and replacing an existing item from the reservoir requires one more random access. The latter may or may not be on the same cache line as the admission state, depending on the reservoir capacity (k) and whether the reservoir's storage is inlined with the admission state.

Admission control takes place for every tuple, so the corresponding time is reduced as the input tuples decrease. Admission, however, happens stochastically with a probability that converges to the sampling rate as more items are considered per reservoir. As a result, admissions are responsible for a small portion of the build time. Furthermore, following the observation that an item will be infrequently replaced, LAQy does not pack the reservoir storage with the admission state. Instead, it stores a pointer to the actual reservoir together with the admission state to reduce the footprint of the hash table data structure used to store the strata.

Figure 3 shows how the build time increases with an increase of the input size. Independently of the number of strata, the sampling time follows a similar trend. However, as the number of strata grows, we observe a higher sampling time even for small input sizes. Each stratum introduces a constant allocation and initialization even if it does not reach its capacity. As a result, while the input size affects the sample building time, strata initialization time, count, and random accesses to stratum state have a significant impact on the sample build time, exacerbated by the total cardinality of QCS.

Figure 4 shows how the build time changes for different admission rates and a number of groups by varying the reservoir capacity k. While the reservoirs are full at the end of the sampling, their capacity has a minor impact on the build time for variations from 0 to 2000 tuples/reservoir over the full data input of 6B tuples. Note that while plotted lines show small increases of tuples per reservoir, the actual increase of admitted tuples is a multiple of the increase in reservoir capacity: if 500 more tuples are accepted to each reservoir, and there are 20 reservoirs, then a total of 10,000 more tuples are admitted overall into the sample. In contrast, the number of reservoirs, or equivalently the number of groups, has a significantly higher effect on the build time: the random access to evaluate whether a tuple should be admitted to a group overshadows the per-reservoir tuple admission.

Key observations & predicate predictability. LAQy builds on top of two observations. First, the number of strata and the number

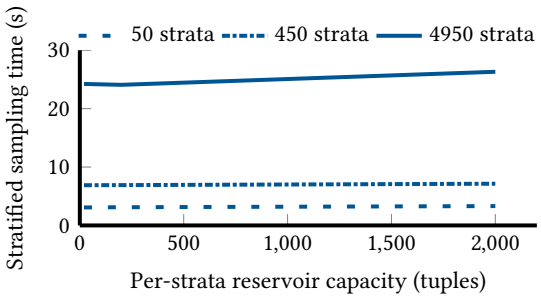


Figure 4: Impact of incrementing the per-reservoir capacity.

```
SELECT C1, SUM(C2) FROM T
WHERE C3 > 5 [AND C1 IN ('G1', 'G2')]
GROUP BY C1
```

Figure 5: Predicated aggregation query example.

of tuples have the highest impact on sampling time, while per-stratum capacity k has a lower impact on the build time. Second, increases in the reservoir capacity have a marginal impact on the overall sampling time while controlling the desired error bounds by obtaining sufficient sample support.

Online AQP approaches [19] rely on the first observation to (conditionally) perform a filter pushdown in the query plan to reduce the input as soon as possible for the following (sampling) operators. Pushing operations below sampling, however, reduces the generality of the sample: the sample is specific to the pushed-down operations, and thus, if materialized, its reuse is limited to queries that subsume the predicates and sample characteristics. BlinkDB [2] makes the differentiation between predicates that are predictable for the given queries and creates predicate-specific samples, where beneficial, to specialize in optimizing to storage and processing budget. For unpredictable predicates but predictable QCSs, pushing filters down below sampling operations significantly decreases the sample usefulness, as the predication can have different predicate values across queries which would have to subsume the existing sample characteristics and predicates [28]. LAQy takes advantage of the two observations above to accelerate sampling for unpredictable predicates by exploiting patterns and overlaps present in the query predications that reduce the penalty of mispredictions in the case of partial predicate matches.

4.2 The Cost of Predicate Unpredictability

Suppose we have the following query that we would like to approximate by creating a sample, where the input relation T can be a base table or a subquery result.

If the predicate values are predictable on `C3` and `C1`, the future queries are assumed to satisfy those conditions. This means that building a stratified sample on QCS={`C1`} with filters pushed down before the sampler can answer the future queries with equal or *stricter*, subsuming predicates[1].

[1]If a sufficient sample size support exists for requested error guarantees after applying the stricter predicate.

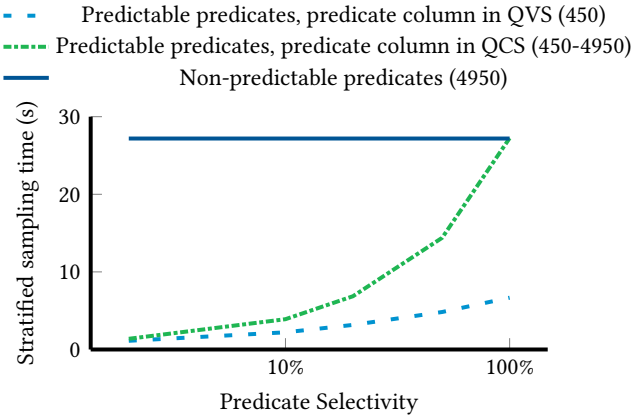


Figure 6: Sampling time for various selectivities.

If the predicate values are not predictable, but predicate columns are, those columns are added to the QCS, without pushing down the filters before sampling. This makes the resulting sample applicable to any predicate on those columns, at the cost of a larger input and more complex stratified sampling with multiple columns in QCS. As the cost of sampling is not negligible at query execution time, the cost of more complex sampling schemes adds a prohibitive penalty as the user may be interested only in limited, yet unpredictable, regions of the dataset, incurring wasteful precomputation at a critical execution path. Suppose this happens if the user is interested only in specific groups in the future queries with a filter on `C1` in brackets.

To demonstrate the impact of selecting one of the above options, Figure 6 shows the sample build time required for creating a stratified sample for different selectivities. Section 7 provides the detailed input and hardware details. We start with a stratified sample on QCS={`C1`} and filter pushdown on `C3` (*predictable predicates on QVS*) which results in 450 strata. To improve the reusability of the materialized sample without any predictions about the runtime predicate values, the column `C3` is added to QCS without any filter pushdown, resulting in 4950 strata (*non-predictable predicates*).

However, when the predicate is on a column `C1` that would participate on the QCS due to being part of the `GROUP BY` clause, the filter can be pushed down along with adding the column to the QCS. In the best case, on the highly selective end for the newly added QCS this approach prunes the sample input instead of making a strict decision to add the entire column to the QCS (*predictable predicate on QCS*).

There is up to 19-24x slowdown to create a non-predicate-specific sample, representing an average slowdown of 6.7-11x across the selectivities to resolve the predicate value unpredictability with the **existing all-or-none sample matching** systems and approaches. The goal is to construct a reusable sample as the highest speedup is achieved using a previously materialized sample (offline AQP) instead of online sampling. The leading cause of the predicate predictability rigidity is the strict requirement of predicate subsumption for reuse. **The sample reuse dichotomy** creates a clear-cut tradeoff between reusability and performance, driven by predictions instead of the actual workload with *overlapping* predicates.

4.3 Relaxing the Sample Predicate Predictability

There is, however, a middle ground between predictable and unpredictable predicates. To demonstrate this, consider the two queries in Figure 1. If the value for predicate on C2 is known ahead of time, the query belongs in the predictable predicate category. If that value varies, then the predicate would be considered unpredictable. Suppose a sample is materialized as a side-effect of workload. In that case, the sample can be reused if the predicates are subsumed [28], assuming enough tuples in the sample satisfy the predicate to achieve the desired error guarantees.

To our best knowledge, existing systems and algorithms opt to rebuild samples if the predicate is not fully subsumed, even if generating an online sample for the missing range $[2, 6)$ would suffice to answer the query using the previously materialized, *effectively offline* sample, as in Figure 1.

We call a sample generated for the uncovered range $(x, y]$ a Δ *(delta)* sample, the process of generating such sample *expansion* as it expands the samples' predicate coverage, and the process of combining the reusable sample with Δ sample *merging*. We call the overall sampling *lazy*, as it defers and relaxes the strict predictability rules as late as possible while reducing the amount of work that samplers need to do.

Summary. Our approach relaxes the predictability requirements to improve the reuse of the samples previously materialized through workload-aware methods described by prior work [2, 28]. We modify the physical sampling algorithm and obtain an equivalent *logical* sample with the requested characteristics. We explore the reservoir-based sampling algorithms (simple reservoir sample, stratified reservoir sampling), commonly used in systems, but mainly since *merging* two or multiple reservoirs preserves the characteristics of the final (reservoir) sample [41]. We propose a method that bridges *offline* and *online* sampling, **complementary** to prior art.

5 Lazy Sampling with LAQy

Traditionally [2, 19], the columns in a stratified sample are split into two sets: the QCS (*Query Column Set*) and the QVS (*Query Value Set*). The QCS contains the columns that affect the participation of the rows in the output, such as columns participating in filters, join conditions, and grouping columns. The QVS has the remaining columns, such as the aggregation columns.

We will briefly describe the main steps of how LAQy relaxes the predicate requirement for sample reuse, depicted in Figure 7:

(1) Starting from an approximable query, an optimizer (e.g. [19, 28]) decides on the *logical* sampler placement (striped green circle). The sampler can have as input base relations or subqueries in a more general case.
(2) The sample store attempts to find a materialized sample that has the requested characteristics of the *logical* sampler.
   (a) If there is a sample that already covers the predicate space with a subsuming predicate, we can use the existing sample if sufficient sample support exists after filtering, fully reusing the *offline sample*.
   (b) If no sample exists overlapping in predicates and requirements, *online sampling* is performed only.
   (c) If a partially overlapping sample exists, calculate the non-overlapping predicate for the Δ query and prepare the sample for merging in the query execution pipeline. Proceed to the next step (3).
(3) The predicates are pushed down the original query plan to initiate online Δ sampling.
(4) Online and offline (stratified) reservoirs are merged, producing the equivalent logical sample.

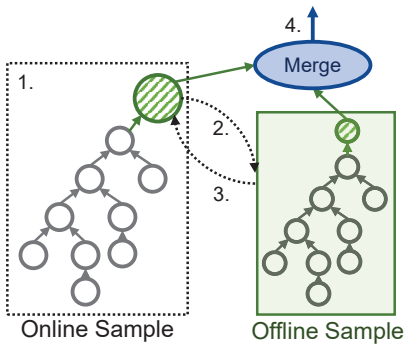


Figure 7: LAQy: the relaxed sample reuse steps.

LAQy combines samples and triggers the generation of a new partial (delta) sample only for the relevant input data not covered by the existing samples. Thus, for each sample, LAQy maintains the *Query Predicate*, *Query Input*, *QCS* and *QVS* that represent the (*logical*) input of each sampler (the striped circle, Offline Sample in Figure 7). Making this information part of the sample description makes explicit that the sample is malleable and reusable, based on specific conditions outlined in the rest of this section.

Algorithm 1: Lazy sampling with LAQy.

Input : Query Q with logical sampler S
Output: Lazy sampler S_{lazy}

$S' \leftarrow$ get existing sample with QCS and QVS of S
if *exists*(S') **then**
  if S' ***subsumes*** *the predicates of* S **then**
    $S_{lazy} \leftarrow S'$ `/* full reuse: offline */`
  else
    if S' ***overlaps*** *with the predicates of* S **then**
      `/* partial reuse: delta range sample */`
      $S_\Delta \leftarrow DeltaSample(S, predicate\Delta(S, S'))$
      $S_{lazy} \leftarrow SampleMerge(S_\Delta, S')$
    else
      $S_{lazy} \leftarrow S$ `/* no reuse: online */`
    end
else
  $S_{lazy} \leftarrow S$; `/* no reuse: online */`
end

To combine two samples, LAQy relies on selecting samples that cover the area of interest but do not overlap. Consider a query predicated on $C3 > 2$. It can be served by combining a sample on $C3 > 5$ with a sample on $C3 \in (2, 5]$. However, combining a sample on $C3 > 5$ with a sample on $C3 \in (2, 7]$ would sample tuples in the $C3 \in (5, 7]$ range twice, introducing a bias towards those intervals – violating the per-reservoir uniformity assumption. To resolve this

issue, LAQy *extends* samples by pushing down the predicates to merge reservoirs on non-overlapping predicates in QVS, reducing the sampler input as a desirable property of Δ samples.

This paper relies on prior art that analyzes and proposes solutions for optimization rules for sampling for online AQP [19], as well as workload-based predictions and sample materializations [2, 28]. We aim to bridge and relax the dichotomy between the two approaches via flexible sample reuse. Therefore, we focus on a solution that synergetically works with the prior art and present and evaluate our contributions within reasonable starting assumptions. At either extreme, our system would run as purely *online* or *offline* sampling-based AQP system. For example, we will assume that an offline sample exists to focus on the behavior and performance of the proposed relaxed reuse via lazy sampling.

5.1 Sample Merging

Reservoir sampling is amenable to updates and maintenance while preserving the requested sample characteristics. We use the property of well-defined *merge* behavior to enable both the independent, data-parallel scale-up execution as well as the key contribution that relies on sample merging. Different data processing operations, such as filtering or stratification, impact the data distribution. Further, partitioning and/or splitting the data for parallelization potentially introduces additional data skew. As a result, using samples created after such operations requires careful consideration to avoid creating biased samples.

LAQy uses the weighted reservoir sampling algorithm [7] to recover a sample equivalent to a full resample of the input. Specifically, during merging, LAQy weighs the elements of the to-be-merged samples based on what proportion of the input they represent. To do so, each created sample keeps the reservoir R and tracks the running sum of (importance) weights w of previously qualifying elements. This allows LAQy to use the exact reservoir weights as it creates and stores samples just-in-time, effectively maintaining the count of the elements as the weight of each qualifying input element is one. Using these (importance) weights, LAQy calculates the new weights associated with every sample to use them further with the weighted reservoir sampling algorithm so that the elements of the merged sample have the same weight as after a full resample.

The key observation is that two independent reservoirs with their associated weights $\{R_1, w_1\}$ and $\{R_2, w_2\}$ can be *merged* to obtain $\{R_m, w_1 + w_2\}$. Intuitively, reservoir R_1 represents w_1 tuples, R_2 represents w_2 tuples, and a reservoir R_m equivalent to sampling the union of original input data of R_1 and R_2 would have combined weights $w_1 + w_2$. To avoid resampling the original input data, we use the existing samples where we adjust the sampling weight from uniform to biased, as in general case weights are different, where elements of R_1 are selected with probability $\frac{w_1}{w_1+w_2}$; where converse holds for elements of R_2 [41]. More formally, to combine reservoirs R_1 and R_2 into R_m, we perform weighted reservoir sampling [7] with R_1 and R_2 and their weights as input to the algorithm. The result is equivalent to having performed reservoir sampling over the combined input while avoiding repeating processing and accessing the original data represented by R_1 and R_2. Therefore, merging itself does not change the properties of the obtained reservoir as it is a reformulation of the inputs to the algorithm amenable to changing the input weights.

Algorithm 2: Reservoir merging pseudo-algorithm.

Input : Reservoirs R_1 and R_2.
Output: Merged reservoir R_m.

if *only single reservoir defined in input* **then**
  $R_m \leftarrow DefinedReservoir(R_1, R_2)$;
end
if *either reservoir array not full* **then**
  $R_m \leftarrow ReservoirSampling(R_1, R_2)$;
else
  $k_1 \leftarrow getResSize(R_1)$; $k_2 \leftarrow getResSize(R_2)$;
  $w_1 \leftarrow getResWeight(R_1)$; $w_2 \leftarrow getResWeight(R_2)$;
  if $k_1 == k_2$ **then**
    $R_m \leftarrow ProportionalSampling(R_1, R_2, w_1, w_2)$;
  else
    $R_m \leftarrow ScaledPropSampling(R_1, R_2, k_1, k_2, w_1, w_2)$;
  end
end
return R_m;

Algorithm 3: Stratified sample merge pseudo-algorithm.

Input : Data pipelines with stratified samples D_1 and D_2.
Output: Merged stratified sample S_m.

$combinedInput \leftarrow Union(D_1, D_2)$;
$keyCols, reservoirs \leftarrow UnpackCols(combinedInput)$;
$S_m \leftarrow GroupBy(key = keyCols, val = reservoirs, aggFun =$ *Algorithm 2*);
return S_m;

In Algorithm 2 we name the weighted reservoir sampling *ProportionalSampling*. Since, in general, even the reservoir sizes k may differ, we introduce *ScaledPropSampling* to further bias the weight by the ratio of $k1$ and $k2$, obtaining the weight factor of $\frac{k_{scaled}}{w}$ to achieve proportional reservoir merge via weighted reservoir sampling of the inputs. Other cases cover the paths where only one reservoir is defined and when one of the reservoirs is still not full and did not enter the probabilistic step of the algorithm. LAQy maintains reservoirs with their weights, and the reservoir sizes are parameters known at runtime.

To generalize stratified sampling with multiple reservoirs, Algorithm 3 represents step 4 (*Merge*) in Figure 7. Strata are represented by stratification keys (*keyCols*) and reservoirs. Next, we perform an equivalent of a GroupBy operation over strata that unioned represent $D1$ and $D2$, where the aggregation function is Algorithm 2 over the reservoirs that share a stratification key. The case of merging two reservoir samples from Algorithm 2 is equivalent to the case of grouping without a key.

5.2 Issuing Δ Samples with Relaxed Predicates

LAQy differentiates between two general types of reuse across the samples: tightening and relaxing the predicates depending on the available materialized samples that may require issuing a Δ sampling query.

5.2.1 Conditional transition to stricter predicates happens when a sample already covers the wanted space, but the predicate of interest is stricter. We can use the existing sample under the condition that samples have enough support after the predicate pushdown to satisfy the accuracy requirements [28]. No Δ sample needs to be issued if the sample meets the condition; otherwise, a complete online sampling must be performed to satisfy the guarantees.

5.2.2 Relaxing predicates requires adding input tuples for predicate values not covered by an existing sample. We issue a Δ query based on the predicates to find the *inverted*, non-overlapping interval.

5.2.3 Combined tightening and relaxing is the case when predicates require tightening of the predicates on an existing *offline* sample and relaxing via executing the Δ sample for the missing interval. However, the sample support (of each stratum) after applying the predicate $\sigma_{predicate}(S)$ is unknown ahead of time, which may impact the (specified) expected error bounds. If a strict qualifying sample support is needed, we can follow the conservative approach: for all the reservoirs/strata that do not have sufficient support an online query is subsequently executed, which performs sampling after the filter pushdown. This validates if the *exact* low (or lack of) support comes from the original data distribution or as the artifact of sampling. We can continue with the query in a less strict setting and report the obtained error bound with the available data. One approach to reducing the probability of insufficient sample support is introducing an oversampling factor $\alpha \geq 1$ to create reservoirs sized $\alpha \cdot k$. This trades-off space for tentatively higher sample reusability in case of strict predicates, similar to how past approaches [2] create samples with different parameters k. This is compatible with LAQy since we show that the effect of higher reservoir capacity is negligible (Figure 4). However, tuning this parameter is out of the scope of this work.

Predicate relaxing allows for building only a Δ sample on the missing value range. LAQy reduces the input to the expensive sampling operations by minimizing the necessary work a Δ sampler needs to do at query time. We reduce the wasteful processing while taking a sampling decision as late as possible to process only the data of interest to the user, maximizing the reuse of previously materialized *online* or *offline* sampling effort.

6 System

We implement LAQy in Proteus [10, 20, 34], a parallel in-memory DBMS engine. Proteus uses LLVM to generate customized code for each query, as in the prior work on JIT engines [10, 25, 27, 36]. As Proteus did not have AQP support, we extended it by (1) introducing sampler operators with the corresponding code generation routines, (2) adding a sample lifetime management module that captures the generated samples to allow reuse on subsequent queries, and (3) implementing sample merging aggregation functions and operators.

6.1 In-Memory Scale-Up Processing

We test our approach using a state-of-the-art analytical engine to demonstrate the practicality and understand the tradeoffs of AQP in modern systems. We analyze the bottlenecks that sampling-based AQP systems will encounter with parallel processing over high-bandwidth, low-latency storage media. We opt for in-memory, scale-up processing to compare the sampling performance with the technological limits (e.g. memory bandwidth) and use the analytical engine performance as the optimized baseline fully utilizing the available hardware. Integrating LAQy with disk-based or scale-out systems [2, 19, 28] is a topic for future research.

6.2 Sampling Operators and Code Generation

To maximize pipelining and reduce the overhead of sampling we introduced sampling as specialized operations in the existing code generation infrastructure. Specifically, we introduced reservoir sampling as a new aggregation function that produces a bag of items. Stratified sampling is then implemented as a group-by that aggregates the input using the reservoir aggregation function. Similarly, the reservoir aggregation function can be used with a reduction to create a simple reservoir sample. Lastly, due to the operator fusion, function calls to the C++ standard random number generation functions introduced a non-negligible overhead, so we used a low-overhead random numbered generator, the Lehmer generator [31], and inlined it with the generated code, allowing the state to be kept in registers.

6.3 Sample Memory Management and Reuse

As we implement sampling as an aggregation function, at the end of the sampling phase, we move the ownership of the generated aggregation result into the sample manager. Specifically, at the end of a stratified sampling phase, multiple reservoirs are ready. Instead of registering and moving each of these reservoirs into the sample manager, we transfer the ownership of the hash-table used by our group-by implementation. The sample manager then serves the hash-table produced by the group-by as a stratified sample to the queries operating on the corresponding stratified sample. This process does not require moving or copying the data. Maintaining the sample across queries has no performance penalty other than the storage space.

We represent reservoirs as the admission control state and a pointer to a contiguous memory region, the reservoir storage, that stores the sampled tuples. This decouples the reservoir storage from the admission control allowing a smaller hash-table footprint during the sample creation phase – enabling a greater access locality when the overall admission rate is low. Furthermore, by carrying the entire state, we can independently merge the reservoirs for both the scale-up processing where samples are combined and collected after an exchange operator[14], as well as for the proposed relaxed sample merging (Figure 7).

7 Evaluation

Hardware setup. We run our experiments on a server with dual-socket Intel(R) Xeon(R) Gold 5118 CPU (2x12 cores, 2x24 threads, HyperThreading enabled) with 384GB DDR4 RAM. Unless otherwise noted, all the experiments run on 48 threads.

Dataset. We use the Star Schema Benchmark [29] data for our experiments, using the scale factor (SF) 1000 in a binary column layout. This yields about 6B tuples in the fact table (`lineorder`), with about 23GB of data per single 4-byte integer column. We add a unique identifier column (`lo_intkey`) to the `lineorder` table as an 8-byte integer value ranging from 0 to the number of elements in the table, randomly shuffled to enable fine-grained selectivity control without implying a specific data ordering. The necessary columns are preloaded in memory for the experiments. We summarize the QCS sizes for the stratified sampling experiments in Table 1.

Table 1: Query column set mapping and |QCS| sizes.

Columns	`lo_quantity`	`lo_tax`	`lo_discount`
Individual |QCS|	50	9	11
1-column |QCS|	50		
2-column |QCS|	450		
3-column |QCS|	4950		

Experiments use 1 to 3 columns to produce up to 4950 strata, defined by the attributes in Table 1. Each stratum is a unique combination of values from stratification columns aligned with the benchmark specification. Our approach is not limited to the specific number of columns or strata. Furthermore, our experimental evaluation is in line with findings in the report from Microsoft's production big-data cluster [18], where authors report that 90% of the input column sets have between 1 and 6 columns. We use a range of strata, further experimenting with selectivities that LAQy uses to reduce the sampling overhead opportunistically, as in Figure 8.

Workload. To evaluate the isolated performance of stratified sampling, we use the query template in Equation (Strat). When testing the sensitivity to QVS selectivity, we use the predicate on `lo_intkey`, and for QCS, we use the predicate on `lo_quantity`. For the sensitivity to the cardinality of QCS and access patterns generated by stratification, we use from 1 to 3 grouping columns, as outlined in Table 1, with $k = 2000$ for reservoir capacity.

```
SELECT GROUPS, AGG(C1, C2) FROM lineorder
[WHERE lo_intkey BETWEEN(lower AND upper)] |
[WHERE lo_quantity BETWEEN(lower AND upper)]   (Strat)
GROUP BY (1/2/3-columns)
```

For the experiments that evaluate the holistic performance of online sampling in LAQy, we use two representative query templates. Equation (Q1) describes a scan-heavy query where the sampler is pushed down to the scan operator. Equation (Q2) describes a query with a random-access pattern due to joins with dimension tables that enable sampling and stratifying on other meaningful dimensions, where the sampler is placed higher in the query plan. For both queries we control and report the selectivity over the fact table (`lo_orderdate`), via the `lo_intkey` attribute.

```
SELECT AGG() FROM lineorder
WHERE lo_intkey BETWEEN(lower AND upper)     (Q1)
GROUP BY lo_orderdate
```

```
SELECT AGG() FROM lineorder, date, supplier, part
WHERE lo_intkey BETWEEN(lower AND upper)
AND s_region="AMERICA" AND p_category="MFGR#12"   (Q2)
AND ... (JOIN) GROUP BY (d_year,p_brand1)
```

To simulate an exploratory workload with changing predicates, we generate two types of query sequences. The first sequence represents a `long-running` analysis: the user is running the specified query template over 50 iterations such that they progressively extend the value range and narrow it down or use the same interval at a specified rate r. The second sequence represents when the user changes the focus of interest during their analysis, where 60 queries are split into 3x20 batches that we name `short-running` analyses, which are similar in setup to the `long-running` sequence.

Whenever randomization is used, we manually set the generator seeds to have repeatable and mutually comparable experiments. We select the starting point uniformly at random in the value interval, use geometric distribution to instantiate the per-query value range around the starting point, and use $r = 0.3$ as the rate when the same or narrower value range occurs.

7.1 Stratified Sampling and Reuse

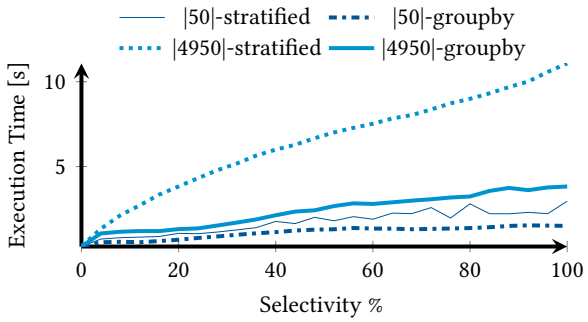
Firstly we evaluate the cost and the potential selectivity-driven savings for the case of stratified sampling. We evaluate in isolation the parameters that affect the sample building time and the time to build the Δ samples by pushing down the predicate on both QVS and QCS as the basis for the relaxed predicate execution of LAQy.

To compare the performance of the stratified sampling operation against the optimized analytical engine baseline, we measure the execution time against the `GroupBy` operator, which shares the data access pattern with stratified sampling. Traditionally, `GroupBy` aggregates the tuples on the fly, while `Stratified Sampling` creates and maintains reservoirs per group/stratum.

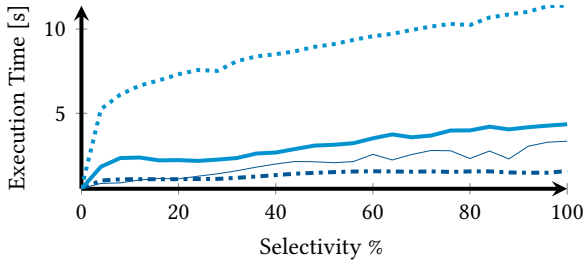
Both GroupBy and Stratified Sampling operations have a random access pattern driven by the cardinality of grouping/stratification keys |QCS|, which impacts the processing time. In Figure 8a we vary the selectivity over |QCS|, with 100% selectivity corresponding to 50 and 4950 groups respectively. On top of the random access pattern, Stratified Sampling further has a processing overhead of the reservoir maintenance algorithm that has to be performed over as many reservoirs as there are groups/strata.

The second factor that impacts the cost of stratification is the total number of tuples processed using a filter that does not directly correlate with the number of strata. We evaluate this by filtering on the column that belongs in the Query Value Set (QVS) instead of QCS. While the total number of tuples reduces the execution time in all cases, the speedup comes from incurring random access patterns defined by the number of groups/strata fewer times (as characterized by selectivity) and not directly impacting the access pattern (Figure 8b). Low selectivities (0%-2%), as shown in Figure 8c, can impact both the number of strata and the overall number of processed tuples.

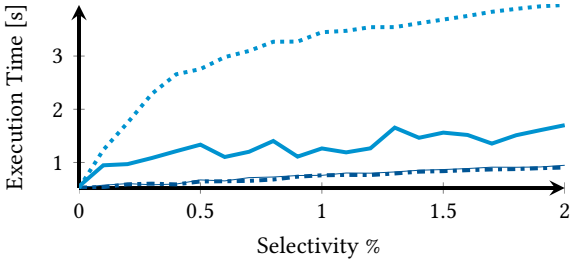
Takeaway. Sampling with relaxed predicate execution through filter pushdown has the potential to reduce the sampling time proportional to the relaxed predicate range (selectivity) that the Δ



(a) Selectivity on the QCS column.

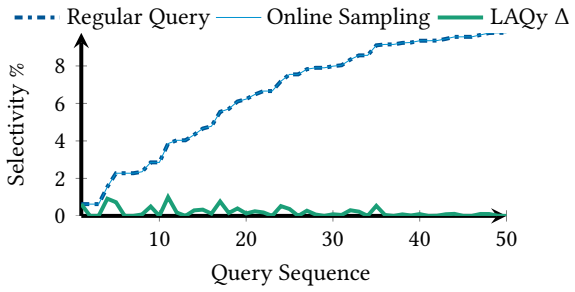


(b) Selectivity on the QVS column.

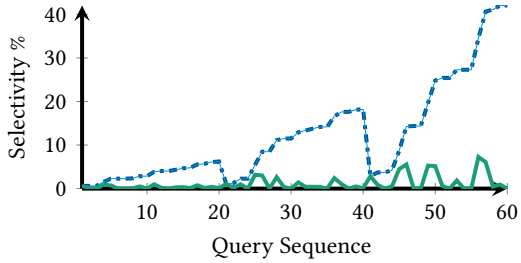


(c) Selectivity on the QVS column - focus on low selectivity.

Figure 8: Stratified sampling and GroupBy time.



(a) Long-running exploration sequence.



(b) Short-running exploration sequence.

Figure 9: Selectivities for the experimental query sequence.

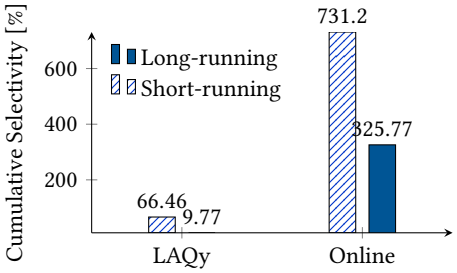


Figure 10: Cumulative selectivities processed in the sequence.

query would request instead of sampling over the entire range. This reduces the cost of online sampling by lowering the unpredictability penalty in a workload-driven lazy-sampling scheme.

7.2 Reuse in Data Exploration Patterns

Exploratory data analysis contains reuse patterns as users will direct and refine the queries based on the previous results and their focus of interest [12]. To showcase possible reuse opportunities, we use two query sequence patterns where users refine and extend the range of values of interest in two settings, as described in the **Workload** paragraph of Section 7.

For both the `long-running` (Figure 9a) and `short-running` (Figure 9b) query sequence, we collect the generated predicate ranges and convert them to input selectivity over QVS of the fact table, which represents the effective input cardinality of the online sampler. Workload-oblivious strategies such as online sampling and (regular) query execution execute on the full specified range. In contrast to that, LAQy uses prior workload as reuse opportunities and results in a lower selectivity range that has to be processed at runtime by issuing a Δ-sample and merging. Furthermore, when the selectivity reaches 0, it indicates that there exists a prior sample that can be used and removes the requirement to perform even a scan over the data.

Observing the above selectivities cumulatively in Figure 10, we show the acute problem of workload-oblivious online sampling. Despite possible reuse opportunities for the suitable query pattern, the same data may be processed many times over, which results in effective cumulative selectivity greater than 100%. LAQy processes only the relevant data whenever possible, processing at most 100% of the data for the qualifying query pattern.

Takeaway. Online Sampling is oblivious to the prior analysis and reuse opportunities, processing and sampling the input as specified by the query. To improve reuse, LAQy detects if an overlap exists for reuse opportunities and effective selectivity reduction.

7.3 Reducing the Cost of Online Sampling

LAQy benefits from reuse opportunities due to overlapping predicates, which enable lazy sampling speedup. In concrete terms, we first present the breakdown of the cumulative processing time of Q1 in Figure 11. First, scan time is lower in the case of LAQy due to

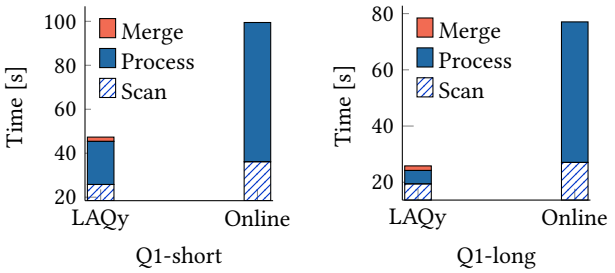
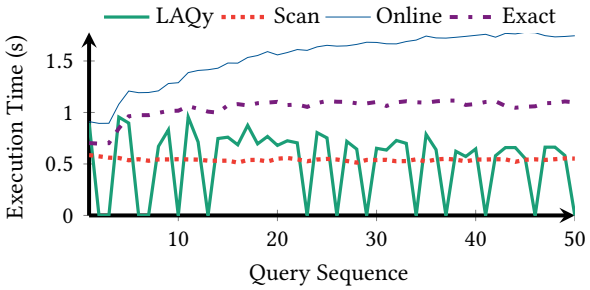
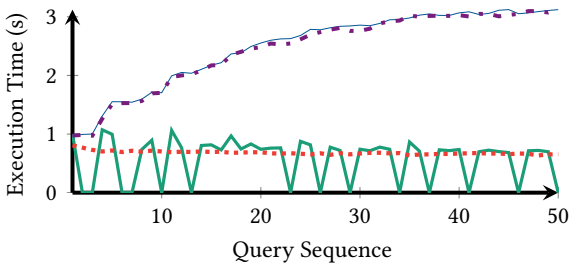


Figure 11: Cumulative processing time breakdown.



(a) Q1 pattern



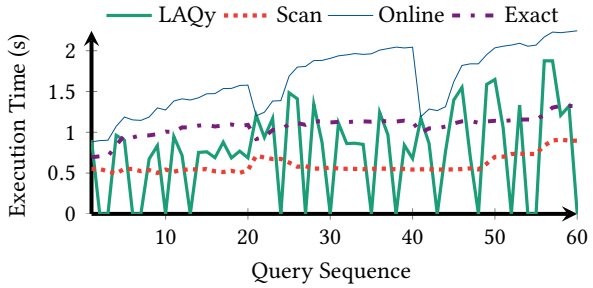
(b) Q2 pattern

Figure 12: Long query sequence, per-query execution time.

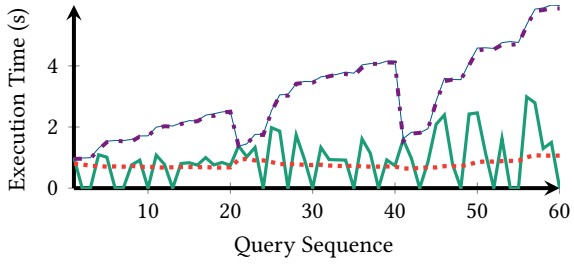
detecting full sample reuse opportunities. Second, the processing time (excluding scan) is lower due to lazy sampling and issuing only required Δ samples. Finally, there is a negligible merge overhead to produce an equivalent sample since merging the reservoirs operates over the data samples.

7.3.1 Long-running sequence: high reuse Lazily sampling in the long-running sequence benefits from high reuse opportunity, due to increasingly low selectivity due to incremental range increase. This enables online sampling to effectively become cheaper than the quivalent exact query with same access pattern (groupBy), and to approach the (ideal) cost for online sampling - which is a simple scan at memory bandwidth, or dip below the memory bandwidth wall when a qualifying (offline) sample exists.

When the sampler is pushed-down to scan, the benefit comes from reducing the input to the sampler (Figure 12a). When the sampler resides after operations such as joins, the benefit comes from both reducing the input to the sampler and all the prior operations (Figure 12b), without affecting the quality of the sampler, as we are effectively sampling past the join operator with reduced input.



(a) Q1 pattern



(b) Q2 pattern

Figure 13: Short query sequence, per-query execution time.

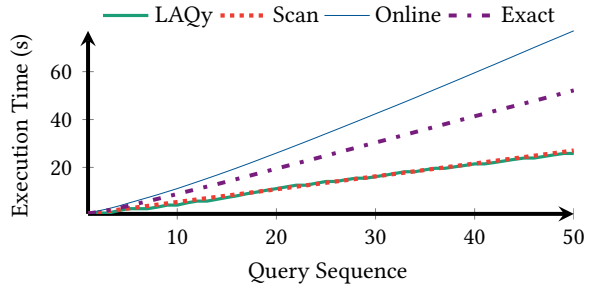
7.3.2 Short-running sequence: adapting to change We analyze the case when users occasionaly completely change the focus of analysis and process new regions of data. Similarly, this could happen if there are multiple linked query dashboards issuing different query patterns as predicate changes.

The initial cold-start queries run in a regular online mode the queries with index 0, 20, and 40 in Figure 13b and Figure 13a. However, onwards, LAQy opportunistically reuses the previously sampled regions of the data based on the logical sampler definition. This ranges from fully reusing a sample, therefore avoiding even the data scan, to creating partial samples and merging them with the existing ones, while not regressing above online sampling cost.
Takeaway. LAQy reduces the cost of online sampling by partial sampling and merging, and benefits from reduced input to the sampler, reduced processing cost of preceeding operators, and reduced scan cost if there exists a full (offline) sample match.

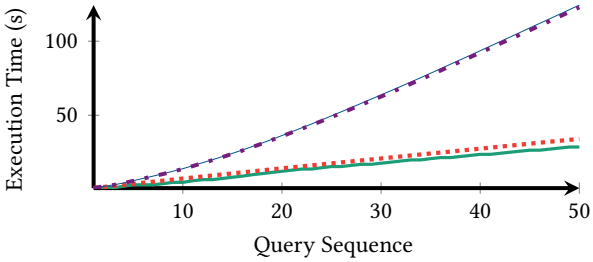
7.4 Efficient & Reusable Sampling with LAQy

In case there is a good workload prediction, offline sampling methods have the potential to create highly reusable set of samples. In case the workload is unpredictable or evolving, our approach reduces the misprediction penalty by creating additional samples only over the queries and data relevant to the user - which is definitely known at runtime. We conclude the experimental analysis of LAQy by observing the big picture of cumulative execution time.

7.4.1 Long-running sequence: high reuse Lazy sampling can achieve the cost which is effectively below the scan time for samplers that are pushed down to the base relations (Figure 14a). This is due to combining offline sample use that does not require a scan and progressively creating missing sample components. Equally, this

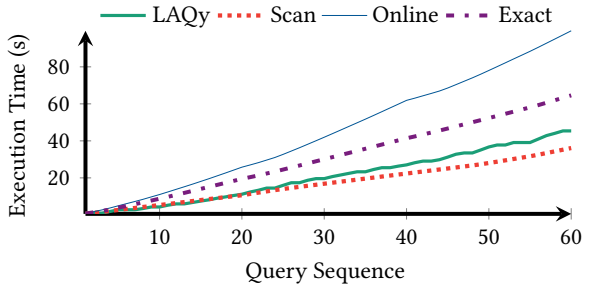


(a) Q1 pattern

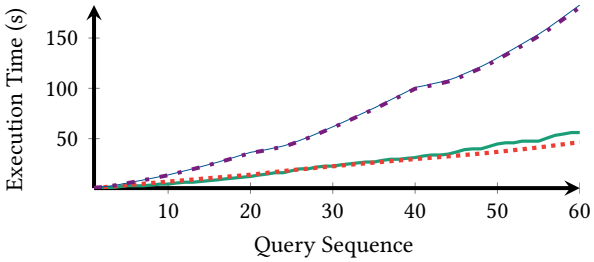


(b) Q2 pattern

Figure 14: Long query sequence, cumulative execution time.



(a) Q1 pattern



(b) Q2 pattern

Figure 15: Short query sequence, cumulative execution time.

could be the case of a good workload prediction where slight predicates deviations occur that our approach can gracefully fix. In a scan-heavy query regular online sampling is slower than the exact counterpart, as it introduces sample processing overhead on top of the input coming in at memory-bandwidth rate (scan).

Similarly, when the sampler is placed further up in the query plan as in Figure 14b, the combined savings from full offline reuse and selectivity-driven partial reuse enable expensive online sampling cost to be reduced to a scan. Online sampling cost is the same as the exact execution, since the input to both operators has a lower throughput due to preceeding joins, where additional online sampling processing is masked with the input data rate.

7.4.2 Short-running sequence: adapting to change When there is moderate reuse opportunity, LAQy enables lazy sampling between the scan and the input rate defined by the preceeding operator(s). In particular, when the sampler is fully pushed down, the lazy sampling enables cumulative execution time comparable to sequential scan (Figure 15a). Otherwise, the cumulative cost of lazy sampling is between the scan and the input rate of input operators, reducing synergetically both the impact of random data access patterns of preceding operations and the sampling cost through delta sampling and predicate pushdown as we show in Figure 15b.

Takeaway. LAQy enables sampling methods to convert the dichotomy of reuse based on full match into a flexible spectrum based on partial matching. As a result, LAQy expands the applicability of existing sampling schemes to a greater query set, with a minimal processing overhead.

8 Related Work

Approximate query processing has been of interest over the years as a method to trade-off accuracy for reduced execution times and has found applications in interactive data exploration [22], data visualization [21], cardinality estimation for query optimization [15], and in novel execution schemes such as speculative execution [38].

The prior work and research in the field of AQP is vast and covers the theory, systems, operators, and approximation schemes [11], well summarized in a survey [23]. We limit the scope of our paper to sampling-based approximations in the context of modern scale-up analytical systems, and we briefly present the related work.

Offline AQP systems. build samples and data summaries ahead of time, based on assumptions of knowing the future workload and static or slowly changing data. Before execution, the sample construction takes place while the system is offline, preparing the relevant samples for use by avoiding access to slower storage mediums and scanning the full data with high latency and I/O bottlenecks; such systems [2, 32] offer significant speedups. However, when an adequate sample is unavailable, fallback to regular execution or online sampling is often necessary, reducing the expected speedups. In recent years ML-based AQP methods [17, 24] have been devised to minimize the storage budget and allow even faster execution times by avoiding data access altogether. However, to achieve this, such systems need to build samples and perform the specified model-based summarizations. By speculating on the workload and spending a specified storage budget, offline AQP systems spend downtime by creating samples and summaries, time in which the user cannot benefit from speedup or using the system. In case of inefficient sampling, this period becomes prohibitively long, resulting in either querying stale data or frequently triggering the sample maintenance procedure. LAQy uses the previously created samples to avoid extensive sample creation and to reduce the size of new samples. As a result, LAQy's continuous sample creation increases the potential of having the required sample ready and

thus brings online approximation methods closer to the benefits of offline ones.

Online AQP systems. [18, 19] perform sampling at runtime and offer speedup through data reduction by injecting samplers at data-intensive parts of the query plan. As the original data needs to be processed, such systems provide lower speedups than their offline counterparts. However, without any workload assumptions, they enable ad-hoc queries and require no storage budget for materializing data summaries. Another flavor of online sampling is Online Aggregation [16], where the query answer is progressively evaluated and estimated. To our knowledge, such systems have a dominant I/O and latency cost [18]. Our work aims to take the flexibility of online AQP methods and make them future-proof and practical by making them efficient in the context of high-bandwidth memory and interconnects inside scale-up analytical engines. Furthermore, they do not take the opportunity of workload patterns that may speed up future queries through sample reuse. In contrast, LAQy's aggressive sample caching methodology reduces the overhead of online sampling to only online sampling for the newly created delta sample.

Hybrid AQP systems. combine online and offline AQP techniques. Taster [28] proposes adapting to the workload by materializing samples relevant to the recent window of queries and otherwise performing online approximations. This is an algorithmic harmonization between online and offline AQP, and it depends on efficient online sampling and fast sample materializations for future use. AQP++ [33] proposes connecting samples with aggregate precomputations such as data cubes. This work is orthogonal to ours, where integrating a lazy sampling scheme would reduce the sampling cost while taking the benefit of using precomputed aggregates to provide faster query responses. Taster makes a coarse-grained decision about full sample matches and does not explore the opportunity of partial sample reuse that would further improve the performance gap between online and offline sampling procedures, especially for the exploratory workloads where the predictability of the useful sample set to precompute is low. Still, the overlap between required (unpredictable) and materialized (predictable) sample sets is high. LAQy exploits this overlap to reduce the sample creation time. Further, it extends the existing techniques with guidelines to update the proposed systems to cover more queries.

Storage management in AQP. Taster [28] and BlinkDB [2] handle sample management based on the available storage. While our evaluations focus on lazy sampling, our approach is compatible with sample-storage-management frameworks to allow workload-sensitive storage provisioning with an allocated storage capacity through a sample expiration policy or evicting it to another storage tier. This remains an interesting research topic for future work.

Model updates and concept drifts. The ubiquitous nature of volatile data has motivated research in machine learning and theoretical approaches to detect and update out-of-date models [13] where new data are monitored to detect when the input data distribution has significantly shifted from the maintained model. When this happens, the online model is retrained or incrementally updated. For example, DriftSurf [39] analyzes the changes in the data distribution to maintain the models. While such works detect and handle changes (drifts) in the input data distribution, LAQy focuses on finding non-sampled input regions informed by the workload. While the two are conceptually related, there are two key differences: 1) the input ranges can reappear very often, and thus query-processing-oriented techniques need fast transitions between old and new concepts; 2) the cost of detecting the input ranges is significantly lower than concept drifts. Specifically, LAQy maintains the samples for each sub-region without fusing them all together, as there is a significant chance that the input ranges may revert towards an older set of input ranges – requiring the sub-samples to construct the samples required for query answering. In contrast, concept drift techniques usually discard/replace the old models, as detecting if a previous model is representative is significantly expensive in online learning. Further, LAQy can rapidly (query-granularity) react to input range changes due to the low overhead associated with detecting the input range, with well-defined properties for updating and merging data summaries [1, 7, 41]. In contrast, concept drifts usually require multiple input points to detect significant (and probabilistic) deviations from the underlying data distribution. As a result, LAQy re-adapts the concept drift ideas to the analytical query processing, where the drifts due to the changing workload are immediately detected and accounted for.

Window-based aggregations. While we focused on changes on the predicates that partition the input ranges, prior work on streams has extensively optimized how to provide fast exact and approximate responses on sliding windows [3, 5, 6, 30, 35, 40].Such streaming approaches often calculate a summary per window slide and then aggregate the summary overlapping with the full window of interest. LAQy builds on this concept by using and aggregating subsamples. However, traditional sliding window approaches 1) do not have to rebalance the subwindow summaries to produce the final summary probabilistically and 2) have a very predictable pattern with respect to the summary that will host the new elements. Despite their differences in query and setting, LAQy can be adapted to such streaming scenarios by adding the time dimension as an additional predication to each sample and using the sample merging techniques to merge samples from different window slides.

9 Conclusion

The reuse dichotomy between the predictable predicates used by offline sampling to speculate on the useful samples and the online sampling, which pays the performance cost of unpredictability in ad-hoc queries, imposes a steep performance penalty, even if an overlapping sample exist. We propose bridging this gap by relaxing the sample reuse requirements and allowing partial sample reuse. We increase the utility of the samples created using the assumption of predictability and pay only the necessary cost to perform online sampling due to the predicate unpredictability. Our lazy sampling approach adapts to the changes in query predicates and improves performance proportionally to the reuse savings. As the data volume continues to grow, to enable faster and interactive analytical queries, judicious sampling becomes an ever-important part of efficient AQP in modern scale-up analytical engines.

Acknowledgments

We thank the reviewers for their insightful comments and suggestions. We further thank our colleagues and friends at the DIAS Lab at EPFL for their valuable feedback.

References

[1] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, 23–34. https://doi.org/10.1145/2213556.2213562

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355

[3] Ran Ben Basat, Seungbum Jo, Srinivasa Rao Satti, and Shubham Ugare. 2021. Approximate query processing over static sets and sliding windows. *Theor. Comput. Sci.* 885 (2021), 1–14. https://doi.org/10.1016/j.tcs.2021.06.015

[4] Altan Birler, Bernhard Radke, and Thomas Neumann. 2020. Concurrent Online Sampling for All, for Free. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3399666.3399924

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[6] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2002. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 215–226. https://doi.org/10.1016/B978-155860869-6/50027-5

[7] M. T. Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* 69, 3 (12 1982), 653–656. https://doi.org/10.1093/biomet/69.3.653 arXiv:https://academic.oup.com/biomet/article-pdf/69/3/653/591311/69-3-653.pdf

[8] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. 2004. Effective Use of Block-Level Sampling in Statistics Estimation. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/1007568.1007602

[9] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 511–519. https://doi.org/10.1145/3035918.3056097

[10] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[11] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1-3 (2012), 1–294. https://doi.org/10.1561/1900000004

[12] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. IDEBench: A Benchmark for Interactive Data Exploration. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1555–1569. https://doi.org/10.1145/3318464.3380574

[13] João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4, Article 44 (mar 2014), 37 pages. https://doi.org/10.1145/2523813

[14] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Rec.* 19, 2 (may 1990), 102–111. https://doi.org/10.1145/93605.98720

[15] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 499–512. https://doi.org/10.1145/3186728.3164145

[16] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. *SIGMOD Rec.* 26, 2 (June 1997), 171–182. https://doi.org/10.1145/253262.253291

[17] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[18] Srikanth Kandula, Kukjin Lee, Surajit Chaudhuri, and Marc Friedman. 2019. Experiences with Approximating Queries in Microsoft's Production Big-Data Clusters. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2131–2142. https://doi.org/10.14778/3352063.3352130

[19] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 631–646. https://doi.org/10.1145/2882903.2882940

[20] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proc. VLDB Endow.* 9, 12 (2016), 972–983. https://doi.org/10.14778/2994509.2994516

[21] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. 2015. Rapid Sampling for Visualizations with Ordering Guarantees. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 521–532. https://doi.org/10.14778/2735479.2735485

[22] Tim Kraska. 2017. Approximate Query Processing for Interactive Data Science. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 525. https://doi.org/10.1145/3035918.3056099

[23] Kaiyu Li and Guoliang Li. 2018. Approximate Query Processing: What is New and Where to Go? - A Survey on Approximate Query Processing. *Data Sci. Eng.* 3, 4 (2018), 379–397. https://doi.org/10.1007/s41019-018-0074-4

[24] Qingzhi Ma and Peter Triantafillou. 2019. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1553–1570. https://doi.org/10.1145/3299869.3324958

[25] Prashanth Menon, Amadou Ngom, Todd C. Mowry, Andrew Pavlo, and Lin Ma. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14, 2 (2020), 101–113.

[26] Robert B. Miller. 1968. Response time in man-computer conversational transactions. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I (AFIPS Conference Proceedings)*, Vol. 33. AFIPS / ACM / Thomson Book Company, Washington D.C., 267–277. https://doi.org/10.1145/1476589.1476628

[27] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[28] Matthaios Olma, Odysseas Papapetrou, Raja Appuswamy, and Anastasia Ailamaki. 2019. 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*. IEEE, 482–493. https://doi.org/10.1109/ICDE.2019.00050

[29] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[30] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. 2015. Sketching distributed sliding-window data streams. *VLDB J.* 24, 3 (2015), 345–368. https://doi.org/10.1007/s00778-015-0380-7

[31] S. K. Park and K. W. Miller. 1988. Random Number Generators: Good Ones Are Hard to Find. *Commun. ACM* 31, 10 (Oct. 1988), 1192–1201. https://doi.org/10.1145/63039.63042

[32] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1461–1476. https://doi.org/10.1145/3183713.3196905

[33] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1477–1492. https://doi.org/10.1145/3183713.3183747

[34] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2043–2054. https://doi.org/10.1145/3318464.3389783

[35] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. 2022. Fast Concurrent Data Sketches. *ACM Trans. Parallel Comput.* 9, 2 (2022), 6:1–6:35. https://doi.org/10.1145/3512758

[36] Viktor Sanca and Anastasia Ailamaki. 2022. Sampling-Based AQP in Modern Analytical Engines. In *DaMoN*. ACM, 4:1–4:8.

[37] Ben Shneiderman. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96)*. IEEE Computer Society, USA, 336.

[38] Panagiotis Sioulas, Viktor Sanca, Ioannis Mytilinis, and Anastasia Ailamaki. 2021. Accelerating Complex Analytics using Speculation.. In *CIDR*.

[39] Ashraf Tahmasbi, Ellango Jothimurugesan, Srikanta Tirthapura, and Phillip B. Gibbons. 2021. DriftSurf: Stable-State / Reactive-State Learning under Concept Drift. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 10054–10064. http://proceedings.mlr.press/v139/tahmasbi21a.html

[40] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *Proc. VLDB Endow.* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940

[41] Chris Wyman. 2021. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Apress, Berkeley, CA, Chapter 22, Weighted Reservoir Sampling: Randomly Sampling Streams, 345–349. https://doi.org/10.1007/978-1-4842-7185-8_22