JANUSAQP: Efficient Partition Tree Maintenance for Dynamic Approximate Query Processing

Xi Liang  
University of Chicago  
xiliang@uchicago.edu

Stavros Sintos  
University of Illinois Chicago  
stavros@uic.edu

Sanjay Krishnan  
University of Chicago  
skr@uchicago.edu

Abstract—Approximate query processing over dynamic databases, i.e., under insertions/deletions, has applications ranging from high-frequency trading to internet-of-things analytics. We present JANUSAQP, a new dynamic AQP system, which supports SUM, COUNT, AVG, MIN, and MAX queries under insertions and deletions to the dataset. JANUSAQP extends static partition tree synopses, which are hierarchical aggregations of datasets, into the dynamic setting. This paper contributes new methods for: (1) efficient initialization of the data synopsis in the presence of incoming data, (2) maintenance of the data synopsis under insertions/deletions, and (3) re-optimization of the partitioning to reduce the approximation error. JANUSAQP reduces the error of a state-of-the-art baseline by more than 60% using only 10% storage cost. JANUSAQP can process more than 100K updates per second in a single node setting and keep the query latency at a millisecond level.

I. INTRODUCTION

Approximate query processing (AQP) studies principled ways to sacrifice query result accuracy for faster or more resource-efficient execution [7], [14]. AQP systems generally employ reduced-size summaries, or “synopses”, of large datasets that are faster to process. The simplest of such synopsis structures are histograms and samples [9], [28], [5], [26], but many others have been proposed in the literature. More complex synopses are more accurate for specific types of queries [41], specific data settings [36], or even are learned with machine learning models [43], [18], [29]. AQP is particularly interesting and challenging in a dynamic data setting, where a dataset is continuously modified with insertions and deletions [14], [30], [3]. In this setting, hereafter denoted as DAQP, any synopsis data structures have to be continuously maintained online.

As an example use-case, consider a database aggregating per-stock order data for the NASDAQ exchange [1]. Suppose, that we would like to build a low-latency SQL interface for approximate aggregate queries over the past seven days of order data. On a typical day, there are 25M new orders that correspond to trades that are placed by brokers (up to 70,000 orders in any given second). A decent fraction of these orders are eventually canceled or prematurely terminated, for a variety of financial reasons. Thus, this database is highly dynamical with a large volume of new insertions (new orders) and a small but significant number of deletions (canceled orders). This paper explores such scenarios in internet-of-things monitoring and enterprise stream processing.

Simple synopses like 1D histograms and uniform samples are easy to maintain dynamically. However, such structures are often inaccurate in high-dimensional data and selective query workloads. More complex synopses structures, e.g., [43], [28] can be optimized for a particular instance (dataset and query workload), but are generally harder to maintain online. For example, recently proposed learned synopses require expensive retraining procedures which limit insertion/deletion throughput [43], [18], [29]. Even classical stratified samples may have to be periodically re-optimized and re-balanced based on query and workload shifts [5]. These, expensive (re-)initialization procedures can significantly hurt insertion throughput, and accordingly, almost all existing AQP systems focus on the static data warehousing setting. A notable exception being the AQUA project [3] from 20 years ago. Unfortunately, the existing techniques that are designed for dynamic data, such as sketches and mergeable summaries [13], [4], [36], [9], [13], [4], often cannot handle arbitrary deletions or aggregation queries with arbitrary predicates easily.

Traditional sketches [8], [9] focus on frequency, distinct values, and quantile problems. They do not handle arbitrary range queries with low error guarantees and they only consider updates in the streaming setting, i.e. they do not handle arbitrary insertions and deletions. Our system has a few similarities with high dimensional histograms which is another type of sketching that has been used for aggregation queries. We propose a dynamic method that combines partitioning (histograms) with stratified sampling and extend it to handle dynamic updates efficiently. Multi-dimensional histograms usually cannot be updated easily under arbitrary insertions and deletions, for example [39], [16] do not handle general updates, and [12] does not guarantee optimality. Furthermore, most histograms focus on equi-depth, equi-width or V-optimal partitioning. We maintain a partitioning that guarantees that the confidence intervals we get are close to the optimum (shortest) confidence intervals from the best partitioning.

In addition, mergeable summaries [4], [13] focus mostly on frequency and quantile problems. Traditional mergeable summaries for aggregation queries mainly include (uniform, stratified, reservoir) sampling. While sampling techniques support efficient updates in the database, the error is far from optimum. Our method is essentially a technique that uses mergeable summaries for aggregation queries. However, in contrast to traditional mergeable summaries, our new system uses a more involved technique: Maintain a partitioning over the space and a stratified sampling under arbitrary insertions and deletions. We maintain and combine low-error summaries...
with only a few higher-error summaries to return a better estimation than traditional mergeable summaries techniques.

Thus, it is understood that most synopsis data structures have at least one of the following pitfalls in our desired dynamic setting: throughput, drift, or generality [7].

This paper explores the DAQP problem and studies ways that we can mitigate the pitfalls of prior approaches with a flexible synopsis data structure that can continuously re-optimize itself. We present JANUS AQP, a new DAQP system, which supports SUM, COUNT, AVG, MIN, and MAX queries with predicates under arbitrary insertions and deletions to the dataset. The main data structure in JANUS AQP is a dynamic extension of our recently published work [27], [28], which we call a Dynamic Partition Tree (DPT). DPT is a two-layer synopsis structure that consists of a: (1) hierarchical partitioning of a dataset into a tree, and (2) a uniform sample of data for each of the leaf partitions (effectively a stratified sample over the leaves). An optimizer determines the best partitioning conditions and sample allocations to meet a user’s performance goals. For each partition (nodes in the tree), we calculate the SUM, COUNT, MIN, and MAX values of the partition. Any desired SUM, COUNT, AVG, MIN, and MAX query can be efficiently decomposed into two parts with the structure: a combination of the partial aggregates where the predicate fully covers a partition in the tree, and an approximate part where the predicate partially covers a leaf node (and can be estimated with a sample). More importantly, this structure is essentially a collection of materialized views and samples, which can be maintained incrementally.

A core contribution of JANUS AQP is online synopsis optimization. JANUS AQP continuously monitors the accuracy of all of its DPT synopses to account for data and workload drift. When a synopsis is no longer accurate, it triggers a re-optimization procedure that resamples and repartitions the data. This re-optimization problem is both a significant algorithmic and systems challenge. From an algorithmic perspective, JANUS AQP needs an efficient way to determine the optimal partitioning conditions in dynamic data. We propose an efficient algorithm based on a dynamic range tree index that finds a partitioning that controls the minimax query error (up to an approximation factor). From a systems perspective, re-optimization poses a bit of a logistical challenge. New data will arrive as the new synopsis data structure is being constructed. We design an efficient multi-threaded catch-up processing algorithm that synchronizes new data and historical data without sacrificing the statistical rigor of the estimates.

Overall, the new system JANUS AQP we proposed has the following benefits over previously known indexes for approximate query processing. It handles arbitrary dynamic updates efficiently (comparing to the static indexes PASS [28], VerdictDB [32], or some dynamic histograms [16] that do not handle arbitrary updates), it provides theoretical guarantees on the confidence intervals (comparing to the machine learning based indexes such that DeepDB [18]), the query procedure accesses only a small synopsis of data and does not touch the original data set so the communication throughput is low (comparing to other tree-based indexes such as [23], [24]), and the estimation error is always low (comparing to traditional sketches [9], or histograms for aggregation queries) without making any assumption about the spatial/value-domain distribution of the data (comparing to [26]). Furthermore, our new system maintains the strongest theoretical guarantees on the error comparing to all the other methods we discussed. In particular, the confidence intervals we get are close to the optimum confidence intervals we could have from the best partitioning.

II. BACKGROUND

We first introduce the core concepts behind the synopses used in this work.

A. Dynamic Approximate Query Processing

We assume an initial database table \( D^{(0)} \). This table \( D^{(0)} \) is continuously modified through a stream of insertions and deletions of tuples. As a design principle, we assume that insertions are common but deletions are rare. With each insertion or deletion operation, the table evolves over time with a new state at each time step: \( D^{(0)}, D^{(1)}, \ldots, D^{(i)}, D^{(i+1)}, \ldots \)

A synopsis is a data structure that summarizes the evolving table. For each \( D^{(i)} \), there is a corresponding synopsis \( \Sigma^{(i)}: \Sigma^{(0)}, \Sigma^{(1)}, \ldots, \Sigma^{(i)}, \Sigma^{(i+1)}, \ldots \). In DAQP, the problem is to answer queries as best as possible from only the \( \Sigma^{(i)} \). For a query \( q \), the estimation error is defined as the difference between the estimated result (using the synopsis) and the true result (using the current database state): \( \text{Error}(q, \Sigma^{(i)}) = |\mu(D^{(i)}) - \mu(\Sigma^{(i)})| \). We further assume that there is sufficient cold/archival storage to store the current state of the table \( D^{(i)} \). This data can be accessed in an offline way for initialization, re-optimization, and logging purposes but not for query processing.

In the full version [2] we show that there are a few notable differences of our dynamic setting from the “streaming” setting since we allow arbitrary insertions and deletions.

B. Related work

There is significant research in histograms and their variants that is highly relevant to this project [20], [25], [19]. V-Optimal histograms construct buckets to minimize the cumulative variance [20]. There are works on multi-dimensional histograms [26], and histograms on the streaming/dynamic setting [17], [16]. Like histograms, JANUS AQP constructs
partitions over attribute domains and aggregates within the partition. However, we contribute different partition optimization criteria than typically used in histograms and novel techniques based on geometric data structures to scale partitioning into higher dimensions. Furthermore, our system works in the general dynamic setting, unlike [16] where the number of total items must remain the same. Another related area of research is into mergeable summaries that compute a partition of the data and optimize sampling at a data partition level [37], [27], [4], [13], [36]. The DPT used in JANUS AQP very much behaves like mergeable summaries but it combines low-error statistics over an up-to-date partitioning of the space, along with stratified sampling to improve the error. Furthermore, some prior work mostly focuses on a streaming setting without support for deletion [36]. Similarly, sketches [8], [9] have been used to find a summary of data to answer approximately a variety of queries efficiently. However, they also do not handle arbitrary range queries using space independent of the size of the full database. Mergeable summaries and sketches usually focus on optimizing different types of problems such that frequency queries, percentile queries, etc. This paper shows how to operationalize a general DAQP system for aggregation queries with both systems and algorithmic contributions relating to the design of dynamic synopses and their continuous optimization. Our system can handle arbitrary updates and can estimate any arbitrary predicate query with provable confidence intervals.

In databases, a number of tree-based indexes, such as the improved $R^*$ tree [24], have been used to support range aggregation queries efficiently. The space of such indexes is (super-)linear with respect to the input items. The query procedures need to have access to the entire tree-index that contains the entire dataset. That leads to high communication throughput or high I/O operations comparing to JANUS AQP where the queries are executed in a small synopsis of data stored in a local machine or RAM with zero communication throughput. In another line of work, tree-based data structures are used to return a set of $k$ uniform samples in a query range. More specifically, in [23], [42] authors construct indexes such that given a query range $Q$ and a parameter $k$, they return $k$ uniform samples from the input items that lie inside $Q$. These samples can be used to estimate any aggregation query in the range query $Q$. There are several issues with these indexes in our setting. First, the design of the index in [23] makes their structure inherently static and it cannot be maintained efficiently. Furthermore, the estimation error in both indexes is the same as the error in the simple uniform random sampling schema. In Section VI, we show that the error of our new index in real data sets is always less than half of the error in uniform random sampling, so our new index always outperforms these range sampling indexes. In addition, the communication throughput or the I/O operations during a query procedure of these indexes is a function of $N$, i.e., the size of the input set, so they cannot be used on big data. Finally, the dynamic tree structure in [26] can store a synopsis of data in a tree-based index and use only this synopsis/index to return estimations of a range aggregation queries, which is also the case in our system JANUS AQP. However, there are two main differences. The index in [26] returns a good estimation only if an assumption about the spatial/value-domain distribution of the data is made, while JANUS AQP uses stratified sampling and it always returns unbiased estimators with small error without assuming any distribution over the data. Furthermore, while their partition tree in [26] can handle dynamic updates, its structure/partitioning remains unchanged. In our index we maintain a near-optimal partitioning over the updates. As we show in Section VI-F, running experiments on real data, re-partitioning is essential in order to maintain a small error.

Dynamic AQP problems have been discussed in prior work [14], however, most existing systems have focused on a static data warehousing setting [5]. The Aqua system [3] did consider the maintenance of its synopsis data structures under updates. However, these synopses were relatively simple and only samples and histograms. Furthermore, we discuss systems issues such as catch-up processing that was not discussed in [3] or any subsequent work [15].

Many new AQP techniques use machine learning. The basic ideas exist for a while, e.g., [21], [22]. Recently, there are more comprehensive solutions that train from a past query workload [33] or directly build a probabilistic model of the entire database [18], [43]. We show that these systems are not optimized for a dynamic setting. Even when they can be updated efficiently with warm-start training, their throughput is much lower than JANUS AQP.

C. Partition Trees for AQP

We propose a new dynamic data synopsis and optimization strategy that is an extension of our previous work [28]. In particular, we proposed a system called PASS (which we call SPT for “static partition tree”). SPT synopses are related to works such as [26] in the data cube literature and hybrid AQP techniques [34]. We showed that with appropriate optimization of the partitioning conditions, an SPT could achieve state-of-the-art accuracy in AQP problems.

1) Construction: An SPT is a synopsis data structure used for answering aggregate queries over relational data. To use SPT, the user defines an aggregation column (numerical attribute to aggregate) and a set of predicate columns (columns over which filters will be applied). An SPT consists of two pieces: (1) a hierarchical aggregation of a dataset, and (2) a uniform sample of data for each of the leaf partitions (effectively a stratified sample over the leaves). The system returns a synopsis that can answer SUM, COUNT, AVG, MIN, and MAX aggregates filtered by the predicate columns.

To understand how this structure is useful, let us overview some of its formal properties. A partition of a dataset $D$ is a decomposition of $D$ into disjoint parts $D_1, ..., D_B$. Each $D_i$ has an associated partitioning condition, a predicate that when applied to the full dataset as a filter retrieves the full partition. Partitions naturally form a hierarchy and can be further subdivided into even more partitions, which can then be subdivided further. A static partition tree $T$ is a tree with $B$ nodes (where each node corresponds to a partition) with the following invariants: (1) every child is a subset of its
parent, (2) all siblings are disjoint, and (3) the union of all siblings equals the parent. In an SPT, each node of the tree is associated with SUM, COUNT, MIN, and MAX statistics over the items in D that lie inside the node.

2) Query Processing: Using an SPT, a user can estimate the result of a query as follows. Essentially, the query processing algorithm identifies “fully covered” nodes that are contained in the query predicate and “partially covered” ones that overlap in some way. Exact statistics from the “fully covered” nodes can be used, while estimates can be used to determine the contribution of “partially covered” ones. We present SUM, COUNT, AVG for brevity, but it is also possible to get estimations for MIN and MAX.

Step 1: Frontier Lookup. Given a query predicate q, traverse the tree top-down to retrieve two sets of nodes partitions: R_cover (nodes that fully cover the predicate) and R_partial (nodes that partially intersect the predicate). Nodes that do not intersect the predicate can be ignored.

Step 2: Partial Aggregation For each partition in R_cover, we can compute an exact “partial aggregate” for the tuples in those partitions. For a SUM/COUNT query q: agg = \sum_{R_i \in R_cover} SUM(R_i), for an AVG query, we weight the average by the relative size of the partition: agg = \sum_{R_i \in R_cover} \frac{sum(R_i) \times N_i}{N_q}, where N_i is the size of the partition R_i, N_q is the total size in all relevant partitions of query q, and SUM(R_i) = \sum_{t \in R_i \cap D} t.a is the sum of the aggregation values of all tuples in the partition R_i.

Step 3: Sample Estimation. Each partition in R_partial is a leaf node with an associated stratified sample. Within each stratified sample, we use standard AQP techniques to estimate that partition’s contribution to the final query result [5]. For completeness, we include those calculations here. Suppose a partition R_i has a set S_i of m_i samples and there are N_i total tuples in R_i. We can formulate COUNT, SUM, AVG as calculating an average over transformed attributes: f(S_i) = \frac{1}{N_i} \sum_{t \in S_i} \phi_q(t), where \phi_q(t) expresses all the necessary scaling to translate the samples in query q into an average query population. Let Predicate(t,q) = 1 if tuple t satisfies the predicate of query q, and 0 otherwise, we have

- COUNT: \phi_q(t) = Predicate(t,q) \times N_i
- SUM: \phi_q(t) = Predicate(t,q) \times N_i \times t.a
- AVG: \phi_q(t) = Predicate(t,q) \times \frac{1}{m_i \times \sum_{t \in S_i} \text{Predicate}(t,q)} \times t.a

We run such a calculation for each partition that is partially covered. These results are combined with a weighted combination like before. For SUM/COUNT queries it is: samp = \sum_{R_i \in R_partial} f(S_i). And for AVG queries, it is: samp = \sum_{R_i \in R_partial} f(S_i) \times \frac{N_i}{N_q}. N_i and N_q can be exactly retrieved from the statistics computed for each partition.

Step 4: Final Estimate. The results can be found by taking a sum of the two parts: result = samp + agg. For this result estimate, confidence intervals can be calculated using standard stratified sampling formulas.

PASS and JANUSAQP comparison. As we noted, JANUSAQP is an extension of PASS in the dynamic setting. The main differences and novelties of our new system JANUSAQP comparing to PASS are the following: i) PASS finds a static partitioning that is not changing after insertions and deletions of items. In JANUSAQP we propose algorithms (Subsection V-D) that automatically check if a re-partitioning is needed after the dynamic updates. ii) Even if re-partitioning is allowed in PASS, the algorithms we proposed in [28] do not run efficiently in the dynamic setting. Here we propose dynamic indexes and algorithms with theoretical guarantees that perform much faster than the algorithms in PASS. iii) Even if we use our new dynamic algorithms in PASS, there is no mechanism to compute the exact statistics of the nodes after a re-partitioning happening and there is no mechanism handling the updates as the re-partitioning is executed. JANUSAQP can improve the estimators in the nodes of DPT after a re-partitioning, while handling new dynamic updates and new queries. iv) Last but not least, we implement JANUSAQP on Apache Kafka, so it can be used by real database systems.

III. SYSTEM ARCHITECTURE

A. Construction and Optimization API

First, we overview how users construct synopsis data structures in JANUSAQP. In order to simplify the description of our system, for now, JANUSAQP constructs a different synopsis for each different query template that the user is interested in. In Subsection V-E we describe how JANUSAQP can handle multiple query templates without using additional space. Much like index construction in a database, users choose which attributes to include in the synopsis structure. Each synopsis can answer query templates of the following form:

SELECT SUM/COUNT/AVG/MIN/MAX(A) FROM D
WHERE Rectangle(D.c1, ..., D.cd)

where A is an aggregate attribute and c1, ..., cd are predicate attributes used in some rectangular predicate region (a conjunction of >, <, = clauses). The dimensionality of a synopsis is the number of predicate attributes d. To construct a synopsis, the user must define the following basic inputs:

- **Aggregation Attribute and Aggregation Function.** An attribute A that is the primary metric for aggregation.
- **Predicate Attributes.** A collection of d columns c1, ..., cd that are used to filter the data prior to aggregation.
- **Memory Constraint.** The maximum amount of space that the synopsis can take.

The user can also optionally define a query processing constraint, i.e., the maximum bytes of data that the system should process in answering a query. Finally, the user has the option to set a historical data limit which is how much historical data to include in the synopsis.

Beyond these basic knobs that are relevant to most AQP systems, there are two other considerations discussed here: Catch-Up Processing. Constructing a synopsis will require some amount of computational time. While incremental maintenance might be efficient, constructing the initial synopsis S(0) from the initial database state D(0) might be very expensive if there is a significant amount of initial data. As the initial S(0) is being constructed new data will arrive, and the system...
will require additional processing to catch up. JANUS-AQP optimizes the catch-up process using a multi-threaded system and approximate internal statistics for the partition tree. This process minimizes the amount of time where the system is unable to process new data or queries. The user decides how much processing to expend during catch up, the quicker the system is ready, the higher the error will be.

**Throughput.** The maximum data throughput is the maximum rate of insertions and deletions that the system can support. Throughput depends on the complexity of the synopsis used.

### B. Data and Query API

For processing queries and data, we adopt the PSoup architecture where both queries and data are streams [6]. JANUS-AQP supports three types of requests: insertion of a new tuple, deletion of an existing tuple and querying of the existing tuples. Thus, there are three Kafka topics insert(tupple), delete(tupple), and execute(query).

The use of Kafka, with its timing and delivery guarantees, simplifies the query processing semantics. The system will process the incoming stream of queries in order. Each query will have an arrival time $i$, which is the current database state at the time at which the query is issued. Therefore, we define $q_j^{(i)}$ as the $j$th query in the sequence that arrives at database state $i$. Query results should reflect all of the data that has arrived until the time point $i$.

### IV. DYNAMIC PARTITION TREES

We discuss how Dynamic Partition Trees (DPT) are constructed, how they answer queries, and how they are maintained under updates. Structurally, a DPT is essentially the same data structure as an SPT; however, the way that the partition statistics and samples are represented differ to allow for incremental maintenance. Figure 2 summarizes the basic update process.

#### A. Incrementally Maintaining Nodes

Each node defines a partition and contains statistics (the SUM, COUNT, MIN, and MAX aggregates) of the data contained in that partition. The key challenge is to keep these statistics up-to-date in the presence of insertions and deletions. When an insertion or deletion arrives, an entire path of nodes from the leaf to the root will have to be updated.

**DPT Nodes:** First, we discuss how we represent the statistics in a DPT node. Since the SUM and COUNT are easy to incrementally maintain under both insertions and deletions, we simply store a single SUM and COUNT value for each aggregation attribute. The MIN and MAX values are harder to incrementally maintain. To store the MIN and MAX values, we store the top-k and the bottom-k values in a MIN/MAX heap respectively. The top value of these heaps is equal to the MIN and MAX of all the data in the node.

**Insert New Record:** When a new record is inserted, we start from the root and find the leaf node that the record is contained in. Then, we increment the SUM and COUNT statistics accordingly. Finally for MAX/MIN, we push the new aggregation values onto the heap. If the heap exceeds the size limit $k$, then the bottom value on the heap is removed.

**Delete Existing Record:** When an existing record is deleted, we find the leaf node that the record is contained in. Then we decrement the SUM and COUNT statistics accordingly. Finally for MAX/MIN, if that aggregation value is contained in the heap it is removed from the heap. This might make the heap smaller than $k$. Repeated deletes from the same node might fully empty the heap. We stop removing values from the heap when there is only one value left. When the heap reaches a single element the MIN/MAX estimates received from the nodes are outer approximations where the estimated value is larger than the MAX and smaller than the MIN.

#### B. Maintaining Stratified Samples

Next, we describe how to maintain the samples associated with leaf nodes. We maintain samples using the proportional allocation of stratified sampling. The number of samples in each stratum should be proportional to the size of the stratum over the size of the population. We use a modified version of the well-known technique of reservoir-sampling [40] under updates [15]. Conceptually, each leaf node is associated with a physically disjoint sample of just that partition, i.e., a stratified sample. Instead of implementing physical strata for the stratified sampling, we implement large enough virtual partitions of a single global sample. This global sample can be maintained using a reservoir sampling algorithm and makes it easier to control the overall size of the synopsis under insertions/deletions as well as simplifies concurrency control.

**Sample Representation:** The DPT maintains a “poold sample” (all the relevant samples in a single data structure).
This set of samples has a target size of $2m$ tuples. At the construction time, we choose a set $S$ of $2m$ uniform random samples from $D$. The update procedure ensures that there are always between $m \leq |S| \leq 2m$ samples. The leaf nodes index into this “pooled” sample selecting only the relevant data to their corresponding partitions. Since we get a set of uniform samples, if we make sure that our strata are large enough, the number of samples we get in each stratum satisfies the proportional allocation requirements, up to a factor of 2, with high probability. We formally show it in the full version [2]. This is also the case in the next update operations.

Insert New Record: Suppose we insert a new tuple $t$. If $|S| < 2m$ we add $t$ in $S$. If $|S| = 2m$, we choose $t$ with probability $\frac{|S|}{|D|}$. If it is selected then we replace a point from $S$, chosen uniformly at random, with $t$.

Delete Existing Record: Next, suppose that we delete a tuple $t$ from $D$. If $t \notin S$ we do not do anything. If $t \in S$ then we check the cardinality of $S$. If $|S| > m$ then we only remove $t$ from $S$. If $|S| = m$ then we skip the set $S$ and we re-sample $2m$ items from $D$. As shown in [15] this procedure always maintain a set of uniform random samples. Using a simple dynamic search binary tree of space $O(m)$ we can update the samples $S$ stored in $T$ in $O(\text{height}(T))$ time.

C. Re-initialization and Catch-Up

As we noted before, repeated deletes on the same leaf partition can degrade the accuracy of the synopsis. As we will see in the next section, it is also possible for repeated insertions to degrade the accuracy as well. In such cases, re-initialization of the DPT may be needed where the data structure is re-built and re-optimized over existing data.

Enabling periodic re-initialization is crucial for reliable long-term deployment but is challenging because new data will not simply stop arriving during the re-initialization period. We employ a multi-threaded approach to minimize any period unavailability for processing new data arrival as well as new queries (Figure 3). When re-initialization is triggered, the main processing thread initiates the construction of a new DPT synopsis and the following steps are performed:

1) Optimization Phase (In Parallel)
   - The partition optimization algorithm analyzes the data in the pooled reservoir sample to determine the optimal new partitioning criteria. It returns a new empty DPT with no node statistics.
   - In parallel with (Step 1), the old synopsis is maintained under all insertions and deletions that happen during the optimization algorithm. Queries can still be answered with the old synopsis.

2) (Blocking) Approximate node statistics are populated into the new synopsis using the pooled reservoir sample $S$ (note, that this will reflect any data that arrived during the optimization phase). This is the only blocking step in the re-initialization routine and new data and queries will have to wait until completion.

3) The old synopsis is discarded.

4) The system resamples a uniform sample of data from archival storage to be the new pooled reservoir sample. Queries and results can still be processed on the new synopsis even without a sample.

5) Random samples of historical data are used to improve the node statistics in the background until a user-specified “catch-up” time.

The duration of the catch-up phase can be chosen by the user. For example, in our experiments, the catch-up phase does not stop until we get $0.1 \cdot |D|$ samples. It is worth noting that queries close to the beginning of the catch-up phase will have a higher error, however queries towards the middle or the end of the catch-up phase will have a smaller error. In Section V-D, we describe how to trigger re-initialization. Furthermore, there is only one step (2) where the synopsis is unavailable to process queries and data (has the duration of 100s of milliseconds in our experiments).

D. Answering Queries With a DPT

The query procedure does not access the entire data so the communication throughput (or the I/O operations) is zero. The basic structure of the result estimator is the same as before, especially for the $R_{\text{partial}}$ partitions. However, there are a few key changes due to the nature of the catch-up phase. In SPT, for each partition in $R_{\text{cover}}$, we can compute an exact “partial aggregate” for the tuples in those partitions and combine the partial aggregates. In a DPT, this process changes considering the estimations we get from the catch-up samples. Overall, the estimation of a partition $R_i \in R_{\text{cover}}$ consists of i) estimation using the catch-up samples $H$ and the formulas of Section II-C, ii) the exact statistics of the new inserted tuples in $R_i$, and iii) the exact statistics of the deleted tuples in $R_i$ (recall that the quantities in ii), iii) are stored and maintained as described in the Incrementally Maintaining Statistics in Section IV-A). By taking the sum of i), ii) and subtracting iii) we get the unbiased estimation in partition $R_i$.

Let $H$ be the set of catch-up samples, $H_i \subseteq H$ be the subset of $H$ in partition $R_i$, and $h_i = |H_i|$. All basic notations are defined in Table I. The formulas for estimating COUNT and SUM queries in both $R_{\text{cover}}, R_{\text{partial}}$ from Section II-C contain the factor $\frac{h_i}{m}$ or $\frac{N_i}{m}$, while the formulas for estimating the AVG contain the factor $\frac{N_i}{m}$. In DPT we do not have the exact values for $N_i$. Instead, we use an estimate of the size of the partition $R_i$ denoted by $\hat{N}_i$. In particular we use the catch-up samples $H$ to estimate $\hat{N}_i = \frac{h_i}{N} N$. 

Fig. 3. JANUS-AQP synopses can be re-initialized online using a multi-threaded implementation to minimize unavailability
1) Confidence Intervals: While the estimators do not significantly change from an SPT to a DPT, the confidence intervals are calculated very differently. This is because there are now two sources of errors: estimation errors due to the stratified samples and estimation errors in the node statistics. Both these sources of errors have to be integrated into a single measure of uncertainty. Assuming that all partitions are large enough, the central limit theorem can be used to asymptotically bound the estimation error for SUM/COUNT/AVG queries. Informally, the central limit theorem states that this asymptotic error is proportional to the square-root of the ratio of estimate variance and the amount of samples used $\sqrt{\frac{\text{var}(\text{est})}{m}}$. We simply have to match terms to this formula for all sample estimates and all node estimates because both are derived from samples.

Error in Node Estimates. First, let’s account for all the uncertainty due to catch-up. Recall that $H$ is the set of catch-up samples we have considered so far and $H_i \subseteq H$ is the samples in partition $R_i$ with $h_i = |H_i|$. We note that we do not store the set $H$ or the subsets $H_i$, instead we only use the new catch-up samples to continuously improve the statistics we store in the nodes. Using the notation in the previous section, we can calculate the catch-up variance $\nu_c$: $\nu_c(q) = \sum_{i \in R_{\text{partial}}} \frac{w_i \text{var}(\phi(H_i))}{m_i}$, where $w_i = \frac{1}{N_i}$ for AVG queries and $w_i = 1$ for SUM/COUNT queries. Calculating $\phi(H_i)$ is straightforward. We simply store additional information that allows us to efficiently calculate the variance. For any node $i$ of $T$ we store $h_i$, $\sum_{t \in H_i} t.a^2$, $\sum_{t \in H_i} t.a$.

Error in Sample Estimates. For a partition $R_i \in R_{\text{partial}}$, let $S_i \subseteq D$ be the set of samples in $S$ that lie in partition $R_i$ and let $m_i = |S_i|$. Like the catch-up variance, we can calculate the sample estimate variance $\nu_s$: $\nu_s(q) = \sum_{i \in R_{\text{partial}}} \frac{w_i \text{var}(\phi(S_i))}{m_i}$. We can calculate an overall confidence interval as: $\pm \sqrt{\nu_c(q) + \nu_s(q)}$, where $z$ is a normal scaling factor corresponding to the desired confidence level, e.g., $z = 1.96$ for 95%. As before, $w_i = \frac{1}{N_i}$ for AVG queries and $w_i = 1$ for SUM/COUNT queries. In the full version [2], we show analytically all formulas for computing the variance under different types of queries.

V. OPTIMAL DPT PARTITIONING

A. Preliminaries and Problem Setup

The partitioning algorithm analyzes the pooled reservoir sample of data to determine how best to partition the dataset. The goal of the partitioning algorithm is to find a partitioning such that the subsequent queries issued to the DPT have low error. Given a set of $O(m)$ samples $S$, and a query template the goal is to construct a data structure that supports the following operations. (i) Insert or delete a sample from $S$ efficiently, and (ii) when a partitioning request comes, it creates a near-optimum partition tree $T$ in $o(m)$ time.

Let $Q$ be a set of possible aggregate queries with a predicate. And, let $\Theta$ be the set of all DPT synopses with $k$ leaf nodes. Each DPT consists of a hierarchical rectangular partitioning having $k$ partitions/buckets in the last level. The main optimization we need to solve to find a good enough partitioning is to minimize the maximum error over the query workload: $\min_T \Theta \max_{q \in Q} \text{Error}(q, T)$. The error is defined as the length of the confidence interval, as defined in the previous section. Since the catch-up variance is usually extremely smaller than the sample estimate variance, we focus on minimizing the maximum length of the confidence interval with respect to the sample estimate variance $\nu_s(\cdot)$. Indeed, the catch-up phase is running in the background retrieving a large number of samples as the time passes. Hence, the estimations in every node of the DPT are constantly improving reducing $\text{var}(\phi(H_i))$ for each partition $R_i$. On the other hand, the number of samples $S$ that we are storing explicitly are limited making the sample estimate variance larger than the catch-up variance. For simplicity, when we say variance we always mean the sample estimate variance.

Instead of looking over all possible queries to minimize the maximum error, one only needs to focus on single partitions to ensure they do not have “high-variance” sub-partitions (see full paper [2]). Indeed by considering only these sub-partitions we can still get a $\sqrt{k}$-approximation for COUNT and SUM queries over the optimum partition considering all queries (for $k$ leaves). The approximation factor improves to $\sqrt{2}$ for $d = 1$. For AVG queries the error of the optimum partition of this simplification is the same with the maximum error considering every possible query. The error of a query $q$ inside a leaf node (partition) $R_i$ is defined (expanding the equations from the definition of $\nu_s(\cdot)$) as

$$\frac{N_i^2}{m_i^2} \left| \sum_{t \in q} t.a^2 - \left( \sum_{t \in q} t.a \right)^2 \right|,$$

for SUM/COUNT and AVG queries, respectively. Thus, the optimization problem reduces to finding partitions that do not contain a high-variance “rectangle” of data.

Given a rectangle $R$, the goal is to find a rectangular query within $R$ with maximum variance among all possible queries in $R \cap S$. For now, we assume that we have a dynamic index $M$ with near-linear space such that given a query rectangle $R$, it returns a query $q$ within $R$ with $\nu_s(q) \geq \frac{1}{2} \text{Var}(R)$, for an approximation factor $\gamma > 1$, in $O(M)$ time, where $\text{Var}(R)$ is the variance of the maximum variance rectangular query in $R$. Let $\text{M}(R)$ be the variance of the query returned by the index $M$. We describe this index with more details in Subsection V-C and the full version [2].

B. Partitioning for $d = 1$

Now, we discuss how to solve the partitioning optimization problem in one dimension. We present results for SUM and AVG queries. COUNT can be thought of as a special case of SUM with binary data. The basic trick is to search over a discrete set of possible variance values. For each value $c$, we try to construct a partitioning of $k$ partitions such that in each bucket the length of the longest confidence interval of a query is at most $c$. By systematically reducing $c$ in each iteration, we control for the worst-case error.

Bounding the Error. We assume that the aggregation value of any item in $D$ is bounded by a maximum value $U = O(\text{poly}(N))$ and a minimum non-zero value $L = \Omega(1/\text{poly}(N))$. In the full version [2] we show that in this case, the length of the longest confidence interval is bounded by $O(\text{poly}(N))$ and $\Omega(1/\text{poly}(N))$. 578
Description of Algorithm. We describe the partitioning algorithm for SUM queries. The procedure is identical for AVG queries. For a parameter \( \rho \in \mathbb{R} \) with \( \rho > 1 \), let \( E = \{ \rho^i \mid i \in \mathbb{Z}, \frac{1}{2} \leq \rho^i \leq N(t) \cup \{0\} \) be the discretization of the range defined by the lower and upper bound of the longest confidence interval (as defined in the previous paragraph). We run a binary search on the values of \( E \). For each value \( e \in E \) we consider, we try to construct a partitioning of \( k \) partitions such that in each partition the length of the longest confidence interval of a query is at most \( e \). If there exists such a partitioning we continue the binary search with values \( e' < e \). If there is no such a partitioning we continue the binary search with values \( e' > e \). In the end, we return the last partitioning that we were able to compute.

It remains to describe how to check if a partitioning with \( k \) buckets (intervals) with maximum length confidence interval at most \( e \) exists. A high level description of the algorithm is:

1) For \( i = 1 \) to \( k \)
   a) Let \( b_i \) be the \( i \)-th bucket with left endpoint \( t_a \)
   b) Binary search on samples \( t_j \) to find the maximum bucket \( b_i \) with error at most \( e \)
   c) If \( \sqrt{\mathcal{M}[t_{a,i},t_{j}]} \leq e \)
      i) Continue search for values \( > j \)
      ii) Else Continue search for values \( < j \)
2) If the partitioning contains all samples construct \( T \) using \( b_i \) as its leaf nodes. Otherwise \( T = \emptyset \).

Correctness. In the full version [2], we show \( \sqrt{\mathcal{V}(b')} \leq \sqrt{\gamma \mathcal{M}(b')} \leq \sqrt{\gamma e'} \leq \rho \sqrt{\gamma \sqrt{\mathcal{V}(b')}} \), where \( b' \) is the bucket with the longest confidence interval in the returned partitioning, \( e' \) is the smallest value in \( E \) such that \( \sqrt{\mathcal{V}(b')} \leq e' \), and \( b* \) is the bucket of optimum partitioning with the largest confidence interval. For \( d = 1 \) we have that \( \gamma = 4 \) for SUM and AVG queries respectively, so we get a partitioning where the maximum error is within \( 2\sqrt{\gamma} \) of the optimum error for SUM queries and within \( 2\sqrt{\gamma} \) of the optimum error for AVG queries.

Running time. Since, \( \mathcal{L}, \mathcal{U} \) are polynomially bounded on \( N \) we have that \( |E| = O(\log N) \) and it can be constructed in \( O(\log N) \) time. The binary search over \( E \) takes at most \( O(\log N) \) steps. We can decide if there exists a partitioning with error \( e \) in \( O(kM \log m) \) time. Overall, for constant \( \rho \), the running time of our algorithm is \( O(kM \log m \log N) \).

In the full version [2], we have that in 1-dimensional \( M = O(\log m) \) for SUM and AVG queries.

C. Partitioning in Higher Dimensions

1) Indexing To Find Maximum Variance: We describe the core index \( \mathcal{M} \) that we use in all our partitioning algorithms for any dimension \( d \geq 1 \). The exact description of the index depends on the type of aggregation queries we focus on.

For SUM/COUNT queries, we propose a simple index to find the query with the largest variance in a query rectangle. In particular, we build a dynamic range tree on \( S \). Given a query rectangle \( R \), we split it into two smaller rectangles \( R_1, R_2 \) such that \( |R_1 \cap S| = |R_2 \cap S| = |R \cap S|/2 \). Using a dynamic range tree [10] we return the rectangle \( R_i \) (either \( R_1 \) or \( R_2 \)) with the largest variance. We can show that \( \nu_*^d(R_i) \geq \frac{1}{4} \mathcal{V}(R) \).

The running time and the update time is \( O(\log^d m) \). For AVG queries, the algorithm proposed in [28] cannot be extended to the dynamic case. Hence, we propose a new more involved dynamic index with a better approximation factor. Due to space limitations, all the details and technical proofs are shown in the full version of the paper [2]. We use two dynamic range trees to find rectangles containing a few samples with large sum of aggregation values. If \( q^* \) is the rectangle returned by \( \mathcal{M} \), we can show that \( \nu_*^d(q^*) \geq \frac{1}{4 \log^{d+1} m} \mathcal{V}(R) \). The exact complexities depend on the dynamic range tree structure we use; our data structure has roughly \( O(m \log^d m) \) space, \( O(\log^{d+1} m) \) update time, and \( O(\log^{2d} m) \) query time.

2) Partitioning: We construct a partitioning by building a k-d tree using the dynamic procedure \( \mathcal{M} \) as we described above. Here, we use our improved index \( \mathcal{M} \) to construct a k-d tree. The high level description of the algorithm is:

1) Max Heap \( C \) containing partition \( R_1 \) covering \( \mathcal{D} \)
2) For \( j = 2 \) to \( k \)
   a) Extract partition \( R_j \) with maximum \( \mathcal{M}(R_j) \) from \( C \)
   b) Create a partitioning of \( \mathcal{R} \) of two partitions \( R_{i_1}, R_{i_2} \) by splitting on the median of \( R \)
   c) Insert \( \mathcal{M}(R_{i_1}), \mathcal{M}(R_{i_2}) \) in \( C \)
   d) Set \( R_{i_1}, R_{i_2} \) as children of \( R_i \) in \( \mathcal{T} \)

We can show that such a tree construction returns a partitioning which is near optimal with respect to the optimum partition tree construction following the same splitting criterion: split on the median of the leaf node with the largest maximum variance query. Overall we construct a data structure that can be updated in \( O(\text{polylog} m) \) time. For a (re-)partition activation over a set \( S \) of \( m \) samples, we can construct a new \( T \) with the following guarantees: For COUNT/SUM queries, \( T \) can be constructed in \( O(k \log^d m) \) time with approximation factor \( 2 \sqrt{\beta} \). For AVG queries, \( T \) can be constructed in \( O(\log d m) \) time with approximation factor \( \frac{\log (d+1)}{2} \).

D. Re-Partitioning Triggers

Assume that the current partitioning is \( \mathcal{R} \) and let \( \mathcal{M}(\mathcal{R}) \) be the (approximate) maximum variance query with respect to the current set of samples \( \mathcal{S} \). JANUSAOQP first checks the number of samples in each bucket (leaf node) of the current \( \mathcal{T} \). If there is a leaf node \( i \) associated with partition \( R_i \) such that \( |S_i| \ll \frac{1}{\alpha} \log m \) (where \( \alpha \) is the sampling rate) then there are not enough samples in \( u \) to make robust estimators. Hence, we need to find a new re-partitioning. Even if the number of samples in each bucket is large our system might enable a re-partitioning: For a partition \( R_i \) in the leaf node layer of \( \mathcal{T} \) let \( \mathcal{M}_i = \mathcal{M}(R_i) \) be the (approximate) maximum variance at the moment we constructed \( \mathcal{T} \). Let \( \beta > 1 \) be a parameter that controls the maximum allowable change on the variance. It can either be decided by the user or we can set it to \( \beta = 10 \). Assume that an update occurred in the leaf node associated with the partition \( R_i \). After the update we get \( \mathcal{M}_i' = \mathcal{M}(R_i) \). If \( \frac{1}{\alpha} \mathcal{M}_i \leq \mathcal{M}_i' \leq \beta \mathcal{M}_i \), then the new maximum variance in partition \( b_i \) is not very different than before so we do not trigger a re-partition. Otherwise, the maximum variance in bucket \( b_i \) changed by a factor larger than \( \beta \) from the initial maximum \( \mathcal{M}_i \). In this case a re-partitioning might find a new tree with smaller maximum
error. We compute a new partitioning \( \mathcal{R}' \) and hence a new tree \( \mathcal{T} \). If \( \mathcal{M}(\mathcal{R}') < \frac{1}{2} \mathcal{M}(\mathcal{R}) \) then we activate a re-partition restarting the catch-up phase over the new tree \( \mathcal{T} \). On the other hand, if \( \mathcal{M}(\mathcal{R}') \geq \frac{1}{2} \mathcal{M}(\mathcal{R}) \) then our current partitioning \( \mathcal{R} \) is good enough so we can still use it. Of course, the user can also manually trigger re-partitioning. For example, the user can choose to re-partition once every hour, day, or after \( \tau \) insertions and deletions have occurred. In the full version [2], we also describe how JANUS\textsubscript{AQP} can execute either partial or full re-partitioning.

E. Discussion: Selection of Parameters and Single Synopsis

Given a memory constraint, our system defines the parameters \( m, k \), i.e. the number of samples and the number of leaf nodes in \( \text{DPT} \). The asymptotic space of our index, as we saw in the previous subsections, is roughly \( O(m+k) \), skipping \( \log m \) factors. However, \( k \ll m \) so the asymptotic space of our system is \( O(m) \). In particular, in our experiments we observed that choosing \( k \approx \frac{\log m}{\log \log m} \) always gives a low space and efficient data structure with low error estimates. Hence, using the memory constraint, the space of our index with respect to \( m, k \), and the relation between \( m, k \) our system can automatically set these parameters. If the user also gives a query processing constraint, we might change the values of \( m, k \) to satisfy this requirement. In particular, the query time of our index is bounded by \( O\left(k + \sum_{R_i \in \mathcal{R}_{\text{repart}}} \log^d m_i\right) = O(k \log^d m) \).

So far, we consider that the user defines different synopses for different query templates. Any combination of user’s inputs (aggregation attribute, aggregation function, predicate attribute) creates a different template. We propose two ways to handle multiple templates. The first one has the same theoretical guarantees with respect to the maximum confidence interval as we had for a single synopsis. In the second one, we present a simpler heuristic way to handle multiple query templates. We can still have confidence intervals but we do not have any guarantee over them.

First method: Recall that a synopsis consists of the partition tree \( \text{DPT} \) and a set of at most \( 2m \) samples \( S \). As we described previously the overall space of a synopsis is roughly \( O(m + k) = O(m) \). We describe a simple method having one global set \( S \) of at most \( 2m \) samples and for each query template we only need to construct a different partition tree \( \text{DPT} \). Hence, if the user is interested in \( L \) different query templates the total space of our index is \( O(m + L \cdot k) \). The main idea is that we store \( S \) only once in a dynamic range-tree or a k-d tree and construct one partition tree \( \text{DPT}_h \) with \( k \) leaf nodes for each query template \( h \leq L \). Each node of every \( \text{DPT}_h \) stores the (updated) statistics and the associated hyper-rectangle of the corresponding \( h \)-th partitioning. The only difference now is that we do not explicitly store the samples in the leaf nodes of each tree. Instead, whenever we need access to the samples in a leaf node, we run a reporting query in the range tree (or k-d tree). Finally, we notice that the user can have an overall memory constraint, and not one memory constraint for each different synopsis. Using the overall upper bound, our system can automatically decide and update the number of samples we store or the size of the partition trees that we maintain in order to satisfy the memory constraint.

Second method: There is also a simpler, heuristic way to handle queries from different templates using only one partition tree \( \text{DPT} \). Imagine we construct a partition tree using the SUM aggregation function over the aggregation attribute \( A \) and predicate attributes \( c_1, \ldots, c_d \). If the user asks a query with a different type of aggregation function, say AVG or COUNT, we can still use the same tree to answer the query as long as we store and maintain the sum and the count in each node of the tree. If the user asks a query with a different aggregation attribute \( B \) instead of \( A \) there are two ways to handle it:

i) We store and maintain statistics in \( \text{DPT} \) for all attributes including \( B \) (we always assume constant number of attributes). Hence, we can still answer a query using the partition tree straightforwardly.

ii) If we do not store and maintain statistics for all attributes, then we can perform stratified sampling using the strata and the samples in the leaf nodes of the \( \text{DPT} \). Finally, assume that the system encounters a new query having different predicate attributes. There are a few ways to handle it:

i) Assume that the dynamic partition tree \( \text{DPT} \) we constructed at the beginning was constructed over all possible predicate attributes of the tuples. We can query such tree with any query range over any subset of predicate attributes and return an estimation with confidence intervals.

ii) Simply apply uniform sampling using the samples \( S \), or iii) re-partition the tree for the new predicate attributes.

VI. Experiments

We run our experiments on a Linux machine with an Intel Core i7-8700 3.2GHz CPU and 16GB RAM. We generate query workloads of 2000 queries by uniformly sampling from rectangular range queries over the predicates. We then initialize a JANUS\textsubscript{AQP} instance with a user-specified sample rate, a catch-up ratio and a number of leaf nodes of the partition tree to compare with other baselines (these parameters directly control the Throughput, Query Latency, and Storage Size).

1) Datasets: Intel Wireless dataset. The Intel Wireless dataset [35] contains 3 million rows of sensor data collected in the Berkeley Research lab in 2004. Each row contains measurements like humidity, temperature, light, voltage as well as the date and time each record was collected.

New York Taxi Records dataset. The New York City Taxi Trip Records dataset [38] contains 7.7 million rows of yellow and green taxi trip records collected in January 2019. Each record contains information about the trip including pickUpDateTime, dropOffDateTime, tripDistance, dropOffLocation, passengerCount, etc.

NASDAQ ETF Prices dataset. The NASDAQ Exchange Traded Fund (ETF) Prices dataset[31] contains 2166 ETFs traded in the NASDAQ exchange from April 1986 to April 2020. There are 4 million entries in the dataset and each entry contains the date, the volume of transactions of an ETF on the date, and 4 prices: the price of an ETF when the market opens and closes; the highest and the lowest of its daily price range.

2) Metrics and ground truth: We report the wall-clock latency and the throughput, i.e. number of requests (query/data) processed per second. To measure the accuracy of the system, we report the 95 percentile of the relative error which is the
difference between ground truth and estimated query result divided by the ground truth. We observed that the results we get are similar if we use median or mean relative error. We define the ground truth to be w.r.t all the tuples available when the query arrives, i.e. the true results reflect all insertions and deletions up to its arrival point. To make sure our experiments are deterministic, we fix this sequence up-front and ensure they are the same for each baseline.

3) Baselines: All of these baselines are tuned to roughly control for query latency.

**Reservoir Sampling (RS) and Stratified Reservoir Sampling (SRS).** We construct a uniform sample of the entire data set which is maintained using the reservoir sampling algorithm [40]. We use a variant of RS first designed for the AQUA system that handles both insertions and deletions [15]. Unless otherwise noted, we use a 1% sample of data. For stratified reservoir sampling, the strata is constructed using an equal-depth partitioning algorithm.

**DeepDB.** We also compare with a machine learning-based baseline called DeepDB[18]. DeepDB achieves state-of-the-art AQP results in the static setting, and we chose it as a baseline since it has limited support for dynamic data. In our baseline, DeepDB trains on 10% of the data. We set this to be equivalent to the “catch-up” sampling in DPT.

**Dynamic Partition Tree-Only (DPT).** We compare with a baseline of only using a single DPT synopsis without online optimization. This synopsis is constructed once and then used for the duration of the experiment. Unless otherwise noted there are 128 leaf nodes in a balanced binary tree, the leaf nodes are associated with 1% samples of their respective strata, and the catch-up sampling rate 10% of the data.

**JANUS AQP.** Finally, we evaluate the full-featured JANUS AQP system. This includes a DPT and also performs re-partitioning if needed. Unless otherwise noted there are 128 leaf nodes in a balanced binary tree, the leaf nodes are associated with 1% samples of their respective strata, and the catch-up sampling rate 10% of the data.

The storage costs of the baselines on the NYC Taxi dataset given the typical setting (128 leaf-nodes, 10% catch-up rate, and 1% sample rate) are: reservoir sampling baseline takes about 5MB, JANUS AQP and DPT takes about 6MB, a DeepDB baseline trained with 10% of the data is about 60MB.

**A. Accuracy**

We first evaluate the end-to-end performance of JANUS AQP and the baselines on a 1d problem (1 attribute). For the NYC Taxi dataset, we use the pickUpTime attribute as the predicate attribute and the tripDistance attribute as the aggregate attribute; for the ETF dataset, we use the volume attribute as the predicate attribute and the close attribute as the aggregate attribute; for the Intel Wireless dataset, we use the time and light attributes as predicate and aggregate attribute respectively.

We start with 10% of the data in Kafka which is used by the baselines for initialization (simulating historical data). We incrementally add 10% more data in increments (simulating new data arrival). After every 10% increment, we re-train the model for DeepDB and re-initialize the DPT used by JANUS AQP. Due to space limitation, we report results when 20%, 50%, and 90% of the rows from each dataset are inserted into the system. The median relative error and the corresponding average query latency can be found in Table II.

We can see that JANUS AQP has the overall best accuracy while controlling for query latency. We note that the accuracy of DeepDB is stable as a function of progress. This is because as a learned model DeepDB has a roughly fixed resolution of the data (it does not increase the number of parameters as more data is inserted). These findings are consistent with results from [28]. The accuracy of RS and SRS improves at a cost of a higher query latency.

**B. Online Performance**

We populate Kafka with the first p percent of the NYC Taxi dataset (p varies from 10 to 90). Like before, we initialize JANUS AQP on the first 10% of data and then incrementally add increments of 10% more. In this experiment, we construct a mixed update workload of both insertions and deletions. On the left plot of Figure 4, we show the throughput of handling insertions and deletions using a pool of 12 threads. We can see the performance of JANUS AQP is quite stable and does not change with the size of existing data or the amount of data that have been processed. For each insertion and deletion, we simply find the target node in $O(\log(k))$ and modify the summary. Even though a larger reservoir size increases the overhead of manipulating the samples for reservoir sampling, the increased overhead is unnoticeable. This is because the stratum stored in each node is $\frac{1}{c}$ of the reservoir, and each stratum is independent with others and race condition only happens if two workers are working on the same node.

On the right plot of Figure 4, we show the re-optimization time cost in sec. by JANUS AQP and DeepDB. The cost to initialize JANUS AQP increases with the number of tuples stored in Kafka but it is still much cheaper than DeepDB. It is worth noting that the re-optimization cost of DeepDB is the cost of re-training instead of incremental training. This is mostly due to the constraint of the API exposed by DeepDB, and we observe that re-train a model with $2n$ samples is faster than train a model with $n$ samples then incrementally train another $n$ samples. The results suggest that complex, learned synopses are not ideal in the dynamic setting. In [2] we study the performance of JANUS AQP on deletions.

**C. The Catch-up Phase**

1) Accuracy: We use the entire Intel wireless dataset as the existing data. We compare a set of JANUS AQP (128, c, 1%) instances where the catch-up goal c varies from 1% to 10% with a step of 1%. When each JANUS AQP instance reaches the catch-up goal, we use it to evaluate the same set of 2000 random queries generated using the light attribute as the aggregate and the time attribute as the predicate attribute.

The results can be found in the left plot of Figure 5. As a reference, we also show the accuracy of an RS baseline with 1% sample rate. We notice that JANUS AQP (128,1%,1%) has no advantage against the RS baseline because neither the

---

1 We omit the results of DeepDB on the ETF dataset in Table II due to large error (> 1000%) for SUM queries. The error of COUNT queries is reasonable.
samples nor the summaries built during catch-up could provide better accuracy. As we increase the catch-up ratio, we can see an improvement in accuracy because the quality of the summaries built by the catch-up phase improved. Comparing with the expensive offline pre-processing used in [28], we believe the catch-up phase is a better alternative that provides another knob to tune the tradeoff between accuracy and cost.

2) Overhead: The overhead of the catch-up phase comes from two sources: the loading and processing of the samples. Data loading time measures the time spent on calling the Kafka poll() API, transferring the data, and ETL operations that are necessary to prepare the data for JANUS AQP to process. Results can be found in the right plot of Figure 5. We can see that the data processing with a single thread takes less than 1.5 seconds for a catch-up ratio of 10%, which is equivalent to a throughput of processing 160,000 tuples per second. Furthermore, the data loading cost is much higher than the data processing cost and we believe the data loading cost can be further improved by more engineering efforts and techniques such as client-assisted data loading [11].

D. Dynamic Query Templates

Using the NYC Taxi dataset, we demonstrate the robustness of JANUS AQP, implementing the heuristic, as described in Subsection V-E. We explore the error under three dynamic workload settings: different predicate attributes, different aggregation attribute, and different aggregation functions.

First, we consider the case where JANUS AQP is constructed using one predicate attribute but the query uses another attribute as predicate attribute. The results can be found in the left plot of Figure 6: the PickupOverPickup baseline is the case where the query and JANUS AQP both use the PickupTime attribute as predicate attribute; the DropoffOverDropoff baseline is the case where the query and JANUS AQP both use the DropoffTime attribute as predicate attribute; lastly, the DropoffOverPickup baseline is the case where the query uses the DropoffTime attribute as predicate attribute while JANUS AQP is constructed using PickupTime as predicate attribute. We observe that when the two attributes differs (DropoffOverPickup) we have the highest error (but still quite competitive). After a re-partitioning with respect to the new predicate attribute the high accuracy can be restored (DropoffOverDropoff). Even though we drew three curves over the entire progress, we assume that the system first considers PickupOverPickup between [0,1–0.3] (cycles), then DropoffOverPickup between [0.3–0.6] (stars), and finally DropoffOverDropoff between [0.7–0.9] (triangles). We present all values over the three curves to show all the details. This is also the case in all plots of Figure V-E.

Second, we consider the case where JANUS AQP is constructed optimizing over one aggregation attribute but the query uses a different aggregation attribute. We assume that the DPT stores and maintains statistics for both attributes. The result can be found in the middle plot of Figure 6 where we show two baselines indicating whether the aggregation attribute used by JANUS AQP is the same with the aggregate attribute of the query. We can see the accuracy of the two baselines are quite close, which suggests that JANUS AQP is quite robust over the change of the aggregation attribute.

Finally, we study how JANUS AQP performs over different aggregation functions. As we described earlier, SUM, CNT and AVG are all part of the statistics we store and maintain in a partition tree optimized over SUM queries, therefore no additional effort is required to take care of a change in aggregation function. Results in the right plot of 6 suggest all the three aggregate functions can achieve good accuracy, which also implies other aggregate functions such as STDDEV that can be composed using SUM and CNT would also perform well.

E. Multi-dimensional Query Templates

Next, we investigate the performance of JANUS AQP with multi-dimensional queries on the NASDAQ ETF Prices dataset. We randomly generate 2000 queries from a 5-D query

| Table II |
|-----------------------------------------|
| MEDIAN RELATIVE ERROR (%) OF 2000 SUM RANDOM QUERIES AND AVERAGE QUERY LATENCY (MS/QUERY) OVER THREE DATASETS. |

| Approach | Intel (%) | NYC (%) | ETF (%) | Intel (ms/query) | NYC (ms/query) | ETF (ms/query) |
|----------|-----------|---------|---------|-----------------|----------------|----------------|
| JANUS AQP | 0.07 | 0.22 | 0.77 | 0.2 | 0.5 | 1.0 | 0.3 |
| DeepDB    | 1.5 | 1.5 | 2.0 | 1.5 | 1.8 | 2.2 | 1.8 |
| RS        | 2.1 | 2.1 | 2.6 | 2.1 | 2.3 | 2.5 | 2.3 |
| SRS       | 1.3 | 1.3 | 1.7 | 1.3 | 1.5 | 1.8 | 1.5 |

Fig. 4. We evaluate throughput of JANUS AQP for insertions and deletions in multi-threaded mode. We also compare the re-optimization cost with DeepDB.

Fig. 5. Varying the catch-up goal from 1% to 10% of the data, we evaluate the accuracy of JANUS AQP and the time cost of the catch-up phase.
template that uses the volume attribute as the aggregation attribute, the date attribute and the 4 price attributes as predicate attributes. We perform the same workflow as we did in Section VI-A. We first compare the median relative error of JANUSAQP (256,10%,1%) with DeepDB. The results can be found in the left plot of Figure 7. We notice that the accuracy of JANUSAQP is better than DeepDB but the relative error increases for both. This is because multi-dimensional queries are usually more selective. Since queries are generated using the entire dataset, we notice that many of the ground truths generated using the first 20% of the data are 0s. Therefore, in the experiment, we start with 30% of the data. On the right plot of Figure 7, we can find the re-optimization cost of JANUSAQP is lower than DeepDB but is more expensive than in the 1D setting. While the increase of dimensions can indeed make it more expensive to process the samples we fetched during catch-up, we believe the re-optimization cost can be further improved with more engineering efforts.

E. Re-partitioning

In the first experiment, using the NYC Taxi dataset, JANUSAQP performs a periodic re-partitioning after every 10% insertions. For comparison, the DPT baseline does not perform any re-partitioning and we evaluate the accuracy. We deliberately skew the insertions by sorting on pickupTimeOfDay so that new insertions would hit a small number of partitions. The results are illustrated in Figure 8(left), we can see the relative error of DPT increases drastically due to a partition tree that becomes more and more imbalanced with new insertions. With periodic re-partition, JANUSAQP keeps the accuracy at a controlled level.

In the second experiment, we use the pickupTimeOfDay as the predicate attribute. Because the dataset is randomly distributed over the pickupTimeOfDay attribute, the insertions are not skewed as in the previous setting. To demonstrate a situation where a re-partition is triggered by deletions, we randomly choose 10% of the nodes and randomly delete half of the samples that belong to these nodes then we insert the next 10% data. After the insertion, the re-partition will be triggered for JANUSAQP. For comparison, we use a DPT baseline that does not perform any re-partition. We perform the same operations to the leaf nodes of the DPT baseline then we evaluate the same set of queries. The results can be found in the right plot of Figure 8, we can see the relative error of DPT increases due to the imbalanced partition tree while the error of JANUSAQP drops because of re-partition.

G. A More Efficient Partitioning Algorithm

In Section V, we propose a binary search-based (BS-based) partitioning algorithm for 1 dimension that is much more efficient. In this experiment, we compare the accuracy and time cost of the BS-based algorithm with the dynamic programming-based partitioning algorithm used by PASS on the Intel Wireless dataset. We implement the BS-based algorithm in Python in our code base of PASS for a fair comparison. We measure the time cost in seconds of each partitioning algorithm given different number of partitions, we also compare the median relative error of the PASS variation over 2000 randomly generate queries. The result can be found in Table III. We vary the number of partitions from 16 to 128, as we increase the number of partitions, the sample size used by the algorithms also increase. We notice that the time cost of the DP-based algorithm increase drastically with the number of partitions while the time cost of the BS-based algorithm increase slightly. On the accuracy side, the DP-based algorithm does lead to a lower error but the BS-based algorithm also introduces good accuracy.

![Figure 6](image-url) Fig. 6. We evaluate the robustness of JANUSAQP over three scenarios where the predicate attribute, the aggregation attribute, and the aggregation function used by the query workload might change from the ones that are used to construct JANUSAQP.

![Figure 7](image-url) Fig. 7. We compare the median relative error and the re-optimization cost of JANUSAQP with DeepDB on multi-dimensional queries.

![Figure 8](image-url) Fig. 8. We compare the accuracy of JANUSAQP and DPT in two scenarios that cause imbalanced partition trees.

| TABLE III |
| COMPARISON OF OUR BS ALGORITHM WITH THE DP ALGORITHM. |
|---|---|---|---|
| Partition Time (s) | DP | BS | 16 | 32 | 64 | 128 |
| Median RE (SUM) | DP | BS | 0.2% | 0.1% | 0.07% | 0.05% | 0.02 |
| Median RE (AVG) | DP | BS | 0.2% | 0.1% | 0.08% | 0.05% | 0.15% |
REFERENCES

[1] Nasdaq bookviewer, 2021.
[2] Full paper. https://arxiv.org/abs/2204.09235, 2022.
[3] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In Proc. of 25th Intl. Conf. on Very Large Data Bases, Citeeseer, 1999.
[4] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems, pages 23–34, 2012.
[5] 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 Proceedings of the 8th ACM European Conference on Computer Systems, pages 29–42, 2013.
[6] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. The VLDB Journal, 12(2):140–156, 2003.
[7] S. Chauburu, B. Ding, and S. Kundula. Approximate query processing: No silver bullet. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 511–519, 2017.
[8] G. Cormode. Sketch techniques for approximate query processing. Foundations and Trends® in Databases. NOW publishers, 2011.
[9] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. Foundations and Trends® in Databases, 4(1–3):1–294, 2011.
[10] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer, 3rd edition, 2008.
[11] C. Dung, D. Tang, X. Liang, A. J. Elmore, and S. Krishnan. Ciao: An optimization framework for client-assisted data loading. In 2021 IEEE 37th International Conference on Data Engineering (ICDE), pages 1979–1984, IEEE, 2021.
[12] D. Donjerkovic, R. Ramakrishnan, and Y. Ioannidis. Dynamic histograms: Capturing evolving data sets. In Proceedings of 26th International Conference on Data Engineering, pages 86–86. IEEE Computer Society, 2000.
[13] E. Gan, P. Bailis, and M. Charikar. Coopstore: Optimizing precomputed summaries for aggregation. Proceedings of the VLDB Endowment, 13(12):2174–2180, 2020.
[14] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In VLDB, volume 10, pages 645927–672356, 2001.
[15] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance with complex queries. In Proceedings of the VLDB Endowment, volume 8, number 12, pages 1461–1476, 2015.
[16] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. Trans. on Data Syst., 31(1):396–438, 2006.
[17] B. Hölprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepspl: Learn from data, not from queries! VLDB Endowment, 2019.
[18] H. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. ACM SIGMOD Record, 30(2):223–234, 2001.
[19] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In VLDB, volume 98, pages 24–27, 1998.
[20] C. Jermaine. Robust estimation with sampling and approximate pre-aggregation. In Proceedings of 2003 VLDB Conference, pages 886–897. Elsevier, 2003.
[21] R. Jin, L. Glmcher, C. Jermaine, and G. Agrawal. New sampling-based estimators for OLAP queries. In 22nd International Conference on Data Engineering (ICDE’06), pages 18–18. IEEE, 2006.
[22] S. Joshi and C. Jermaine. Materialized sample views for database approximate joins. Proceedings of the VLDB Endowment, 20(3):337–351, 2008.
[23] M. Jurgens and H.-J. Lenz. The t/sub a*-tree: an improved r*-tree with materialized data for supporting range queries on olap-data. In Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat. No. 98EX1310), pages 186–191. IEEE, 1998.
[24] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 196–204, 2000.
[25] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. ACM sigmod record, 30(2):401–412, 2001.
[26] X. Liang, Z. Shang, S. Krishnan, A. J. Elmore, and M. J. Franklin. Fast and reliable missing data contingency analysis with predicate-constraints. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pages 285–295, 2020.
[27] X. Liang, S. Sintos, Z. Shang, and S. Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In Proceedings of the 2021 International Conference on Management of Data, pages 1129–1141, 2021.
[28] Q. Ma, A. M. Shangooshhabad, M. Almasi, M. Kerzmann, and P. Triantafillou. Learned approximate query processing: Make it light, accurate and fast. In CIDR, 2021.
[29] M. Olma, O. Papapetrou, R. Appusamy, and A. Ailamaki. Taster: self-tuning, elastic and online approximate query processing. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 24–249. IEEE, 2019.
[30] O. Onyschuk. Stock market dataset. https://www.kaggle.com/dsd141665, 2020.
[31] Y. Park, B. Mozafari, J. Sorensen, and J. Wang. Verdictdb: Universalizing approximate query processing. In Proceedings of the 2018 International Conference on Management of Data, pages 1461–1476, 2018.
[32] Y. Park, A. S. Tajik, M. 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, pages 587–602, 2017.
[33] J. Peng, D. Zhang, J. Wang, and J. Pei. Aeqp+ connecting approximate query processing with aggregate precomputation for interactive analytics. In Proceedings of the 2018 International Conference on Management of Data, pages 1477–1492, 2018.
[34] W. H. Peter Bodik et al. Intel wireless dataset. http://db.csail.mit.edu/labdata/labdata.html, 2004.
[35] R. Poepsel-Lemaitre, M. Kiefer, J. von Hein, J.-A. Quian-Ruiz, and V. Markl. In the land of data streams where synopses are missing, one framework to bring them all. 2021.
[36] K. Rong, Y. Lu, P. Bailis, S. Kundula, and P. Levis. Approximate partition selection for big-data workloads using summary statistics. Proceedings of the VLDB Endowment, 13(12):2174–2180, 2020.
[37] N. Taxi and L. Commission. New york city taxi trip records dataset. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2019.
[38] N. Taxi and L. Commission. New york city taxi trip records dataset. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2019.
[39] N. Taxi and L. Commission. New york city taxi trip records dataset. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2019.
[40] J. S. Vitter. Random sampling with a reservoir. ACM Transactions on Mathematical Software (TOMS), 11(1):37–57, 1985.
[41] B. Walenz, S. Sintos, S. Roy, and J. Yang. Learning to sample: Counting with complex queries. Proceedings of the VLDB Endowment, 13(3):390–402, 2019.
[42] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. Proceedings of the VLDB Endowment, 9(3):84–95, 2015.
[43] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. Proceedings of the VLDB Endowment, 13(3):279–292, 2019.