Approximate Queries over Concurrent Updates

Congying Wang
University at Buffalo
cwang39@buffalo.edu

Nithin Sastry Tellapuri
University at Buffalo
ntellapu@buffalo.edu

Sphoorthi Keshannagari
University at Buffalo
skeshann@buffalo.edu

Dylan Zinsley
University at Buffalo
dylanzin@buffalo.edu

Zhuoyue Zhao
University at Buffalo
zzhao35@buffalo.edu

Dong Xie
The Pennsylvania State University
dongx@psu.edu

ABSTRACT

Approximate Query Processing (AQP) systems produce estimation of query answers using small random samples. It is attractive for the users who are willing to trade accuracy for low query latency. On the other hand, real-world data are often subject to concurrent updates. If the user wants to perform real-time approximate data analysis, the AQP system must support concurrent updates and sampling. Towards that, we recently developed a new concurrent index, AB-tree, to support efficient sampling under updates. In this work, we will demonstrate the feasibility of supporting real-time approximate data analysis in online transaction settings using index-assisted sampling.

PVLDB Reference Format:
Congying Wang, Nithin Sastry Tellapuri, Sphoorthi Keshannagari, Dylan Zinsley, Zhuoyue Zhao, and Dong Xie. Approximate Queries over Concurrent Updates. PVLDB, 16(12): 3986 - 3989, 2023.
doi:10.14778/3611540.3611602

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zzy7896321/aqp_demo_public.

1 INTRODUCTION

Approximate Query Processing (AQP) systems are designed to produce approximate answers for complex SQL queries in split seconds over large databases, where full scans are too slow. It is an attractive alternative for those who cannot afford exact queries over large data due to high response time and/or computational cost, but can trade accuracy for efficiency. For example, a data analyst may issue an aggregation query to find the total revenue of sales that satisfy some predicate over billions of business transaction records. Exactly answering the query incurs at least a linear data access cost. As a result, DBMS cannot keep up with the stringent time requirements due to the faster pace of data growth than memory bandwidth [15], let alone the high economical cost of scale-up and scale-out solutions. In contrast, approximate queries can be answered using a sub-linear or constant sized random sample [4, 5], which makes it an appealing (sometimes must-have) solution.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611602

On the other hand, real-world data are usually under constant updates. If the data analyst has to get real-time insights into the updated data, it is essential for the system to support efficient concurrent queries under updates. Despite the promising query efficiency over large data, most existing AQP systems are not usable for that purpose because they do not support concurrent updates and often make assumptions that data updates happen offline and/or in batches. With those assumptions, they can rely on pre-collected offline samples [13] or pre-built indexes for random sampling [4, 9]. If the system needs to work with concurrent updates by performing online sampling (e.g., [7]), it has to sacrifice the sampling efficiency.

The crux of the issue is that the existing data structures optimized for sampling do not support concurrent updates efficiently, while most AQP techniques treat the sampling mechanism as black boxes provided by underlying database systems. There are generally two approaches for random sampling in AQP systems: scan-based sampling and index-assisted sampling. Scan-based sampling [7] can be implemented as a random filter on top of regular table scans, and thus can leverage the existing concurrency control mechanism such as multi-version concurrency control to support very efficient concurrent updates without much modification to the system. However, it requires at least fully scanning the underlying data once. Index-assisted sampling, on the other hand, can scale linearly to the sample size with a logarithmic factor in terms of table size if we use an aggregate B-tree (aka ranked B-tree [12]), making it a more desirable alternative to scan-based sampling However, concurrent updates in a naïve aggregate B-tree must obtain an exclusive lock in order to update the internal aggregates along a tree path atomically, and thus do not scale well with concurrency in practice .

Towards that, we recently designed AB-tree [17], an aggregate B-tree based on B-link tree [8] for efficient concurrent updates and sampling. In AB-tree, most updaters, with the exception of structural modification operations (SMO), do not block concurrent samplers. That means, AB-tree does not have additional blocking compared to a regular B-link tree and thus can support concurrent update and sampling operations very efficiently. Our implementation in PostgreSQL shows AB-tree indeed achieves similar scalability compared to the standard B-link tree in PostgreSQL.

In this demonstration proposal, we present pgAQP, a PostgreSQL extension that supports approximate aggregation queries with random sampling. Through that, we will show that AB-tree opens the door for supporting real-time approximate query processing in online transaction processing settings. In particular, we introduce a new SQL `TABLESAMPLE` operator in pgAQP called `SWR` (i.e., Simple random sampling With Replacement), which can draw k independent range samples from a base table using AB-tree. It also provides

a set of query rewriting rules to transform approximate aggregation operators into unbiased estimators and confidence intervals. With that, we will demonstrate a typical real-time data analysis scenario where a data analyst issues ad-hoc approximate queries against a database with online transactional updates and the random sampling queries from other users at the same time. We also developed a web interface that allows the audience to customize the ad-hoc query and the background workload, view the accuracy of approximate query answers and their running times, as well as monitor the key system performance metrics.

In the rest of this demo proposal, we briefly review the background and related works in Section 2. In Section 3, we describe how pgAQP is implemented as a set of SQL extensions, rewriting rules, and physical operators in PostgreSQL. In Section 4, we will describe a detailed demonstration plan.

2 BACKGROUND AND RELATED WORKS

The goal of an AQP system is to produce approximate answers to aggregation queries with error guarantees. To formally define that, let's first consider a single-table aggregation query:

```
SELECT SUM(e) as Y FROM T WHERE P^r AND P^f;
```

where e is any expression over base table T, P^r is an optional range predicate and P^f is the remaining filter predicates. We denote the exact query answer as Y, which is unknown to the system.

To produce an approximate answer, the system draws a random sample in T from some probability distribution, and computes an *unbiased estimator* $\tilde{Y}$ such that $E[\tilde{Y}] = Y$. The unbiased estimators with simple random sampling with replacement [5] and Bernoulli sampling [7], two commonly used sampling methods[1], are subtly different. For Bernoulli sampling, each tuple $t \in T$ is independently selected with a fixed probability p. If we denote the sample set as S and $e(t)$ as the value of e evaluated over t, the unbiased estimator is $\tilde{Y}_{\texttt{Bern}} = \sum_{t \in S} e(t)/p$. However, Bernoulli sampling always incurs a linear cost because it needs to make an independent random decision on every tuple in the base table. For simple random sampling with replacement, the sample set S is a multi-set where each tuple $t \in S$ is independently chosen from T with some probability p_t[2]. The unbiased estimator can be computed as $\tilde{Y}_{\texttt{SWR}} = \frac{\sum_{t \in S} e(t)/p_t}{|S|}$.

It is usually impossible to produce the relative error $\epsilon_{real} \triangleq \frac{|\tilde{Y}-Y|}{Y}$ without computing the exact answer Y. Instead, most systems provide error guarantees in the form of *confidence interval*, usually denoted as a symmetric range[3] around $\tilde{Y}$ of width 2ε with respect to a confidence level $\alpha \in [0, 1]$, such that $Pr\{Y \in [\tilde{Y}-\varepsilon, \tilde{Y}+\varepsilon]\} \geq \alpha$. Note that there is no guarantee that a confidence interval is a true indicator for accuracy because 1) there are small probabilities $\tilde{Y}$ is not in it; and 2) the confidence interval itself is also often an estimation from a random process.

[1]Another commonly used type of random sampling, simple random sampling without replacement (SWOR), can be treated as simple random sampling with replacement (SWR) when sample size is small [5]. We omit an in-depth discussion on the pros and cons of them because they are beyond the scope of this work and they provide similar guarantees over small samples.

[2]Uniform sampling where all p_t equal $1/|T|$ is most commonly used but weighted sampling (such as measure biased sampling [4], where $p_t \propto e(t)$) can reduce sample size needed for the same error guarantee, if the expression e is known in advance.

[3]While we assume confidence intervals centers around the estimator, some systems may opt for asymmetric ones for better error guarantees.

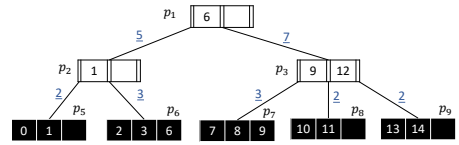


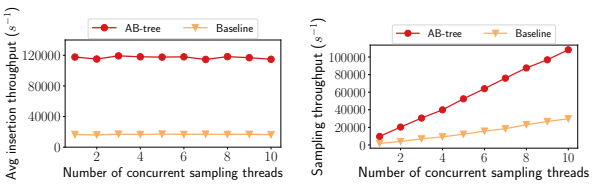
Figure 1: An aggregate B-tree with uniform weights.

Finally, [12] was the first comprehensive study on random sampling in databases. It introduced the ranked B-tree (aka aggregate B-tree, see Figure 1) to support independent uniform random sampling over database tables, which can be easily extended to support independent range sampling and weighted independent range sampling with respect to some weight function $w(t)$. Formally, given a base table T and a range predicate P^r, we want to draw a random sample s from T, such that $\forall t \in \sigma_{P^r} T, Pr(s = t) = w(t)/W$, where $W = \sum_{t \in \sigma_{P^r} T} w(t)$ is the sum of weights of tuples satisfying P^r.

Let's consider full table sampling with $P^r = true$ first. Each child node link in the tree is annotated with the sum of weights in its child node. For instance, the left-most child node link in the root node p_1 has a weight of 5, which is the sum of weights stored in its child node p_2. Suppose we denote all nodes in a particular level as $c_1, c_2, \ldots, c_k$ in index key order (from left to right), their weights as $w(c_i)$ for any $i \in [1, k]$, and define a prefix sum for them as $W_i = \sum_{j=1}^{i} w(c_i)$. Then we can partition the sum range $[0, W)$ into disjoint subranges $[W_{i-1}, W_i)$ whose lengths exactly match $w(c_i)$. If we draw a random number $i \in [0, W)$, it can be mapped to a unique tree path where each node's subrange contains i. Sampling in the tree boils down to finding the subrange that contains i from root to leaf. As the last step, we can sample a tuple from the leaf with probability proportional to its weight. It is obvious that, for any update in this tree, we also need to update all the weights along a tree path from root to leaf *atomically* to ensure the correctness of sampling. Hence, the main challenge of supporting concurrent updates efficiently is the exclusive locking required by atomicity of weight update. For range sampling, we can modify the definition of the weights and ranges such that we only count those that satisfy P^r. Since it is a range predicate, only the weights in left-most path and right-most path that intersects with the *key* range can be different from the stored weights. A known technique is to perform two tree descends to find the two paths and their modified weights and then draw samples per the modified weights. However, this does not work without locking the tree for the duration for sampling because the weights might be changed by concurrent updates.

That said, aggregate B-trees are indeed widely used in many AQP techniques (e.g., [3, 9, 16]) because of it promising run time which scales linearly to sample size with a log factor in table size, and there are many works that seek to improve the efficiency and applicability of sampling indexes in theory [1, 2, 6]. However, to the best of our knowledge, there was no implementation in real-world DBMS because of the difficulty to handle concurrency without excessive blocking. As a result, most AQP systems resort to scan-based sampling, which are either available off-the-shelf (e.g., Bernoulli sampling), or easy to implement in query processing layer [7] or as a middleware [13]. However, they either suffer from long latency due to data scans, or have to use stale offline samples which are not automatically refreshed with data updates.

Our recent work AB-tree [17] tackled problem by using "consistent" weights (i.e., upper bound of weights with relaxed consistency



(a) Insertion throughput (b) Sampling throughput

Figure 2: Scalability test over mixed workload in AB-tree

requirements), which can be updated one at a time rather than atomically. We also showed that weight updates do not have to block tree search and other weight updates by using atomic compare-and-swap over shared-locked B-tree nodes, which is the key to achieve concurrency. In this design, a random number in the range of $[0, W)$ may not map to a tuple because of the relaxation. AB-tree performs rejection sampling to account for that. Note that the rejections induced by the relaxed weight consistency do not introduce excessive rejection because the rejection rate is bounded by the total weight of updates in flight, which is usually a small constant determined by the maximum number of concurrent threads in the system. As a result, AB-tree is able to sustain very high level of concurrent updates and sampling at the same time. Figure 2 shows the insertion and sampling throughput if we run 10 insertion threads and a varying number of sampling threads over a synthetic two-column table with random insertions, where the baseline is an aggregate B-tree with exclusive locks. We refer the readers to [17] for details.

Other related works. There is a rich literature on AQP techniques [10], which usually depend on variants of Bernoulli sampling (aka Binomial sampling), uniform/weighted random sampling with or without replacement [12] (sampling algorithms that improve accuracy such as correlated sampling and stratified sampling [7] can be treated as modified Bernoulli sampling with different, sometimes non-independent, sampling probabilities). There are also AQP systems that combine sampling with pre-aggregation [14] or use machine learning models [11]. While these techniques can improve AQP accuracy, but they are usually more expensive to maintain even without concurrency.

3 PGAQP: A POSTGRESQL EXTENSION FOR APPROXIMATE QUERY PROCESSING

In this section, we describe an experimental PostgreSQL extension, pgAQP, which supports approximate queries on top of the PostgreSQL 13.1 with AB-tree. It supports a subset of approximate SQL queries using simple random sampling with replacement (SWR) and Bernoulli sampling. We utilize the server-side extensibility of PostgreSQL to add the new approximate query operators and to implement query rewriting rules. Specifically, it supports the query syntax in Figure 3, with joins and nested queries.

Currently, there may only be one TABLESAMPLE clause in each query block, which may be either BERNOULLI or SWR. Except for the range variable with table sample clause which must be a base table, all other range variables in the FROM clause may be either nested queries or base tables. Nested queries can also appear in predicates. pgAQP adds a few approximate aggregation operators, including APPROX_COUNT and APPROX_SUM and their Central Limit Theorem based confidence intervals APPROX_COUNT_HALF_CI() and APPROX_SUM_HALF_CI(). Note that the AVG operator is naturally an unbiased estimator for the true average if the underlying sample

```
SELECT <target-list>
FROM T1 TABLESAMPLE {SWR(m)|BERNOULLI(pct)}[, T2, ...]
[WHERE <predicate>]
[GROUP BY <colspec>]
[HAVING <predicate>]
[ORDER BY <orderspec>]
```

Figure 3: Approximate query syntax in pgAQP

is uniform, so we do not have to add an approximate version of that. If any of these approximate aggregation appears in the target list, then pgAQP will rewrite the query plan such that approximate aggregation operators are replaced with unbiased estimators expressed as a mix of standard SQL aggregations and our own UDA (User-Defined Aggregation). For SWR operator, there are some extra works to do during query planning: (1) unlike Bernoulli sampling where sampling probability p is constant, samples in SWR can have different sampling probabilities due to non-uniform weight function or concurrent updates. In fact, the sampling probability of a tuple t can only be computed when AB-tree performs sampling by dividing $w(t)$ stored in the leaf tuple by the total weight of the tree. Hence, we add the sample probability as an extra column in the output in every operator below the aggregation operator. (2) SWR needs to be implemented using index-assisted sampling operator, which does not exist in PostgreSQL. We add it as a customized scan operator. (3) We inject the index-assisted sampling path into the query planner for a base table with AB-tree, if SWR is requested.

Note that, the semantics of SWR is different for aggregation and non-aggregation queries in pgAQP. For the aggregation queries, the sample size m is the number of samples fetched from AB-tree, which are subject to further rejection due to MVCC visibility checks and filter predicate evaluation. Rejected samples are incorporated as 0-valued estimators in aggregation nevertheless. For non-aggregation queries, we retry for each rejected sample until we have m accepted samples, because the user may not be interested in getting a NULL tuple due to rejection and will have to repeatedly issue the same query until getting enough samples. As an illustrating example, let's the following query against the ORDERLINE table in the TPC-C benchmark, where one wants to find a small random sample from all the order lines delivered before a certain date whose amounts are more than 1.5 times of the average amount:

```
SELECT * FROM ORDERLINE TABLESAMPLE SWR(10)
WHERE OL_AMOUNT >=
   1.5 * (SELECT AVG(OL_AMOUNT)
          FROM ORDERLINE TABLESAMPLE SWR(1000))
   AND OL_DELIVERY_D < '2023-03-01'
```

Assuming we have an AB-tree index built on the delivery date with uniform weights, the inner query will perform index-assisted sampling using AB-tree for exactly 1000 samples. Note that while there are no filter predicates in the inner query block, samples can still be rejected due to failed MVCC visibility checks. That said, this sample is already sufficient to produce a quite accurate estimation with a relative error of around 2% for TPC-C data. Then, the query plan generated by pgAQP will use the AB-tree to perform independent range sampling with delivery date before 2023-03-01 and filter the samples based on the estimated value from the inner block. As the samples may be rejected by the filter predicate on OL_AMOUNT, the index-assisted scan will be retried until 10 rows are generated in the outer block. It may also emit a warning if the rejection rate in the outer block is higher than 99.9%, because it indicates the filter predicate could be too selective and yields almost empty results. In

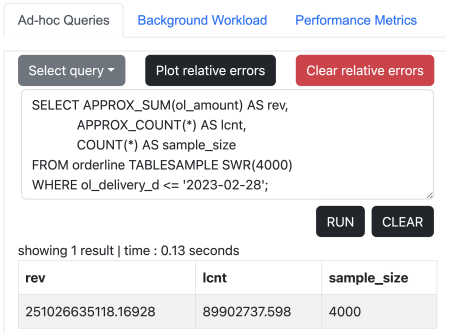


Figure 4: Running ad-hoc queries

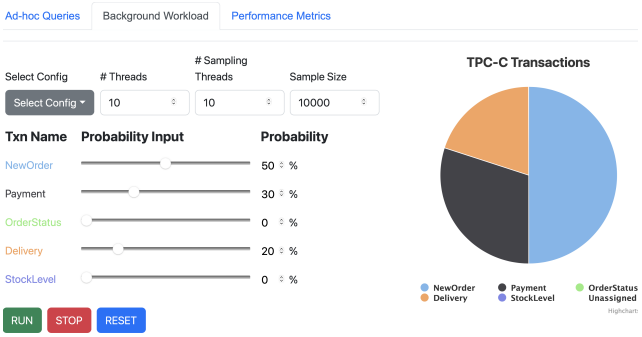


Figure 5: Adjusting background workload

that case, there is no need for sampling and the user may cancel the query to fetch all results for the outer block instead. Another quick note is that this query may be executed even if there are heavy updates in the background, because AB-tree can efficiently support sampling over an MVCC snapshot. Meanwhile, because of its small amount of page reads, it has very limited impact on the transaction processing as well. In contrast, scan-based sampling would take substantially longer to run, and could compete for the limited I/O or CPU resources with the background workload.

4 DEMONSTRATION PLAN

In this demonstration, we aim to showcase the superior performance of index-assisted sampling and the feasibility of supporting highly efficient approximate queries with concurrent online updates. We will simulate an online business transaction database with updates, in which there are also multiple simulated data analysts extracting real-time insights by issuing ad-hoc queries. More specifically, we will set up a TPC-C database with 100 warehouses (27GB, 100 million rows in `orderline` table initially) and a larger one with 400 warehouses (308GB, 2 billion rows in `orderline` initially) in PostgreSQL 13.1 with AB-tree and pgAQP. In addition to the standard primary key indexes, we will also build an AB-tree over the delivery date column in the `orderline` table.

The audience will mainly interact with the system through a web-based frontend interface. In the ad-hoc query panel (Figure 4), they can select from a list of example queries to get familiar with the TPC-C database schema and the approximate query syntax. We prepared 6 queries with 3 variants each: exact query that performs full scans, approximate query using index-assisted sampling (our approach), and approximate query using scan-based sampling (baseline). The audience can also customize the queries. Once a query is executed, the query panel will display the running time and results below it. It can also compute and visualize the relative errors between the ground truth and the estimators. We pre-configured the sample size and sampling rate for both approximate queries such that they draw roughly the same amount of samples and produce estimators with comparable errors.

Once the audience gets familiar with the setup, they can switch to the background workload panel, where they can configure and run a mixed TPC-C and sampling query workload. With the background workload running, the audience can view the performance metrics in the third panel, including AB-tree's sampling throughput, insertion throughput, overall rejection rate, as well as other common system metrics (CPU, memory, I/O). They will be able to study the scalability of the system by varying the number of transaction and sampling threads. At the mean time, they can also switch to the ad-hoc query panel again, and run the queries with the background workload running. They can find out the impact of background workload on three approaches.

ACKNOWLEDGMENTS

Nithin Tellapuri and Zhuoyue Zhao thank Google for a gift for supporting the research.

REFERENCES

[1] Peyman Afshani and Jeff M. Phillips. 2019. Independent Range Sampling, Revisited Again. In *SoCG 2019*. 4:1–4:13.
[2] Peyman Afshani and Zhewei Wei. 2017. Independent Range Sampling, Revisited. In *ESA 2017*. 3:1–3:14.
[3] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2022. Answering (Unions of) Conjunctive Queries Using Random Access and Random-Order Enumeration. *ACM Trans. Database Syst.* 47, 3, Article 9 (aug 2022), 49 pages.
[4] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *SIGMOD '16*. 679–694.
[5] Peter J. Haas. 1997. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *SSDBM '97*. 51–63.
[6] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent Range Sampling. In *PODS '14*. 246–255.
[7] 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 *SIGMOD '16*. 631–646.
[8] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670.
[9] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander Join and XDB: Online Aggregation via Random Walks. *ACM Trans. Database Syst.* 44, 1, Article 2 (Jan. 2019), 41 pages.
[10] 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.
[11] Qingzhi Ma and Peter Triantafillou. 2019. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In *SIGMOD '19*. 1553–1570.
[12] F. Olken. 1993. *Random Sampling from Databases*. Ph.D. Dissertation. University of California at Berkeley.
[13] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *SIGMOD '18*. 1461–1476.
[14] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *SIGMOD '18*. 1477–1492.
[15] Viktor Sanca and Anastasia Ailamaki. 2022. Sampling-Based AQP in Modern Analytical Engines. In *DaMoN '22*. Article 4, 8 pages.
[16] Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. PGMJoins: Random Join Sampling with Graphical Models. In *SIGMOD '21*. 1610–1622.
[17] Zhuoyue Zhao, Dong Xie, and Feifei Li. 2022. AB-Tree: Index for Concurrent Random Sampling and Updates. *PVLDB* 15, 9 (may 2022), 1835–1847.