VerdictDB: Universalizing Approximate Query Processing

Yongjoo Park, Barzan Mozafari, Joseph Sorenson, Junhao Wang
University of Michigan, Ann Arbor
{pyongjoo,mozafari,jsoren,junhao}@umich.edu

ABSTRACT

Despite 25 years of research in academia, approximate query processing (AQP) has had little industrial adoption. One of the major causes of this slow adoption is the reluctance of traditional vendors to make radical changes to their legacy codebases, and the preoccupation of newer vendors (e.g., SQL-on-Hadoop products) with implementing standard features. Additionally, the few AQP engines that are available are each tied to a specific platform and require users to completely abandon their existing databases—an unrealistic expectation given the infancy of the AQP technology. Therefore, we argue that a universal solution is needed: a database-agnostic approximation engine that will widen the reach of this emerging technology across various platforms.

Our proposal, called VerdictDB, uses a middleware architecture that requires no changes to the backend database, and thus, can work with all off-the-shelf engines. Operating at the driver-level, VerdictDB intercepts analytical queries issued to the database and rewrites them into another query that, if executed by any standard relational engine, will yield sufficient information for computing an approximate answer. VerdictDB uses the returned result set to compute an approximate answer and error estimates, which are then passed on to the user or application. However, lack of access to the query execution layer introduces significant challenges in terms of generality, correctness, and efficiency. This paper shows how VerdictDB overcomes these challenges and delivers up to 171× speedup (18.45× on average) for a variety of existing engines, such as Impala, Spark SQL, and Amazon Redshift, while incurring less than 2.6% relative error. VerdictDB is open-sourced under Apache License.

ACM Reference Format:
Yongjoo Park, Barzan Mozafari, Joseph Sorenson, Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA*. ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.1145/3183713.3196905

1 INTRODUCTION

Despite its long history in academic research [68], approximate query processing (AQP) has had little success in terms of industrial adoption [43]. Only recently, a few vendors have started to include limited forms of approximation features in their products, e.g., Facebook's Presto [4], Infobright's IAQ [2], Yahoo's Druid [1], SnappyData [5], and Oracle 12C [67]. While there are several factors contributing to this slow adoption, one of the ways to quickly widen the reach of this technology is to offer a *Universal AQP (UAQP)*: an AQP strategy that could work with all existing platforms without requiring any modifications to existing databases. In this paper, we achieve this goal by performing AQP entirely at the driver-level. That is, we leave the query evaluation logic of the existing database completely unchanged. Instead, we introduce a middleware that rewrites incoming queries, such that the standard execution of the rewritten queries under relational semantics would yield approximate answers to the original queries. This requires that the entire AQP process be encoded in SQL, including the sample planning, query approximation, and error estimation. This approach, therefore, faces several challenges.

Challenges — The first challenge is ensuring statistical *correctness*. When multiple (sample) tables are joined, the AQP engine must account for inter-tuple correlations. Previous AQP engines have relied on foreign-key constraints [8], modifying the join algorithm [39], or modifying the query plan [10, 33]. However, as a middleware, we can neither change the internal query evaluation nor use non-standard join algorithms. With SQL-on-Hadoop systems, we cannot even enforce foreign-key constraints. Thus, we need a different solution that can be implemented by a middleware. The second challenge is the *middleware efficiency*. Pushing the entire computation to the middleware can severely impair performance, because, unlike the database, it is not equipped with query optimization and distributed resources. Finally, there is a *server efficiency* challenge. For general error estimations, previous AQP engines have resorted to computationally prohibitive resampling-based techniques [34, 60], intimate integration of the error estimation logic into the scan operator (e.g., [10, 33, 48]), or even overriding the relational operators altogether [72, 73]. Without access to the query evaluation layer of DBMS, the error estimation has to be expressed as a SQL query, which can be extremely expensive and defeat the purpose of approximation.

Design Criteria — For our UAQP proposal to be practical, it has to meet three criteria. It must offer sufficient *generality* to support a wide class of analytical queries. Despite no access to database internals, it must still guarantee *statistical correctness*, i.e., unbiased approximations and error estimates. Finally, it must ensure *efficiency*. UAQP does not need to be as efficient as a specialized and tightly integrated AQP engine, but to be useful, it still needs to be considerably faster than exact query processing.

Our Approach — First, we sidestep the computational overhead of bootstrap [10, 60] and the intrusive nature of its analytical variants [73] by exploiting the theory of *subsampling* [61]. Note that

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196905

bootstrap's overhead consists of two parts: the cost of constructing multiple resamples, and the cost of aggregating each resample. The traditional subsampling can only reduce the second part—computational cost—by aggregating smaller resamples. Our experiments show, however, the first part—constructing resamples—is still a major performance overhead (Section 6.4).

We thus propose a computationally efficient alternative, called *variational subsampling*, which yields provably-equivalent asymptotic properties to traditional subsampling (Theorem 2). The key observation is that, instead of running the same aggregation query on different resamples, one can achieve the same outcome through a single execution of a carefully rewritten query on the sample table itself. The rewritten SQL query treats different resamples separately throughout its execution by relying on a resample-id assigned to each tuple (Section 4). We also generalize this idea to more complex, nested queries (Section 5).

While integrated AQP engines use hash tables and counters for efficient construction of stratified samples [11, 17, 33], VerdictDB must rely solely on SQL statements to achieve the same goal. However, adjusting the sampling probabilities dynamically (according to the strata sizes) while scanning the data can be extremely expensive. We thus devise a probabilistic strategy that can be implemented efficiently, by exploiting the properties of a Bernoulli process: since the number of tuples sampled per each group follows a binomial distribution, we can (with high probability) guarantee a minimum number of samples per group by adjusting the sampling probabilities accordingly (Section 3.2).

Lastly, unlike most AQP engines that use a single sample for each query [8, 10, 11, 17, 60] (or generate samples on the fly [33]), VerdictDB can choose and combine multiple samples that minimize error (among those prepared offline), given an I/O budget.

Contributions — We make several contributions:

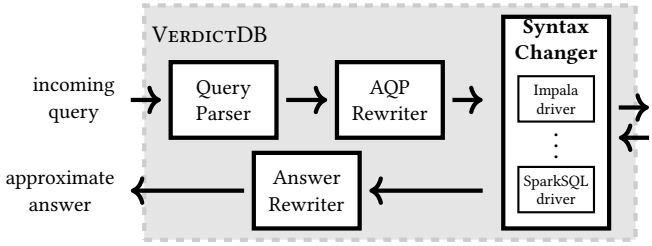
1. We explore the idea of a Universal AQP as a platform-agnostic solution that can work with any database system without any modifications. We propose a realization of this idea, called VerdictDB, that relies solely on a middleware architecture and implements the entire AQP process via re-writing SQL queries.
2. We develop an efficient strategy for constructing stratified samples that provide probabilistic guarantees (Section 3.2). We also propose a novel technique, called *variational subsampling*, which enables faster error estimation for a wide class of queries and can be efficiently implemented as SQL statements (Sections 4 and 5).
3. We conduct extensive experiments on both benchmark and real-world sales datasets using several modern query processors (Impala, Redshift, and Spark SQL). Our results show that VerdictDB speeds up these database engines on average by 57× (and up to 841×), while incurring less than 2.6% error.

While a few other AQP systems have also relied on query-rewriting [8, 9, 24], our techniques enable a wider range of practical AQP queries on modern SQL-on-Hadoop engines (see Section 7 for a detailed comparison).

2 SYSTEM OVERVIEW

In this section, we provide a high-level overview of VerdictDB's components and operations. In Section 2.1, we briefly introduce

(a) VerdictDB acts a middleware between the user and DB.



(b) Internal components of VerdictDB.

Figure 1: VerdictDB Architecture.

VerdictDB's deployment architecture and its internal components. In Section 2.2, we discuss the types of SQL queries that are sped up by VerdictDB. Lastly, in Sections 2.3 and 2.4, we explain VerdictDB's query processing workflow and its user interface.

2.1 Architecture

We first describe how VerdictDB works with other parties (i.e., users and a database system). Then, we describe the internal components of VerdictDB.

Deployment Architecture — As depicted in Figure 1a, VerdictDB is placed between and interacts with the *user* and an off-the-shelf database. We call the database used alongside VerdictDB the *underlying database*. The user can be a data analyst who issues queries through an interactive SQL shell or visualization tool, or any application that issues SQL queries. The user sends queries to VerdictDB and obtains the query result directly from VerdictDB without interacting with the underlying database.

VerdictDB communicates with the underlying database via SQL for obtaining metadata (e.g., catalog information, table definitions) and for accessing and processing data. For this communication, VerdictDB uses the standard interface supported by the underlying database, such as JDBC for Hive and Impala, ODBC for SQL Azure, or `SQLContext.sql()` for Spark.[1] Note that the contributions presented in this work are applicable irrespective of these specific interfaces and can be applied to any database that provides an interface through which SQL statements can be issued. VerdictDB requires the underlying database to support `rand()`, a hash function (e.g., `md5`, `crc32`), window functions (e.g., `count(*) over ()`), and `create table ... as select ...`.

VerdictDB stores all its data, including the generated samples and the necessary metadata, in the underlying database. VerdictDB accesses the underlying database on behalf of the user

[1] `SparkSession.sql()` for Spark 2.0 and above.

aggregates	`count`, `count-distinct`, `sum`, `avg`, `quantile`, user-defined aggregate (UDA) functions
table sources	derived tables or base tables joined via equi-joins; the derived table can be a select statement with or without aggregate functions.
selections (filtering)	expr comp expr (e.g., `price > 100`), expr comp subquery (e.g., `price > (select ...)`), logical `AND` and `OR`, etc.
other clauses	`group by`, `order by`, `limit`, `having`

Table 1: Types of queries that benefit from VerdictDB.

(i.e.,using his/her credentials); thus, VerdictDB's data access privilege naturally inherits the data access privileges granted to its user.

Internal Architecture — Figure 1b shows VerdictDB's internal components. Given a SQL query, Query Parser translates it into logical operators (e.g., projections, selections, joins, etc.). Then, AQP Rewriter converts this logical expression into another logical expression that performs AQP (Sections 3 and 4).

Syntax Changer converts this rewritten logical expression into a SQL statement that can be executed on the underlying database. This is the only module in VerdictDB that needs to be aware of the DB-specific limitations (e.g., no `rand()` permitted in selection predicates in Impala) and its SQL dialects (e.g., quotation marks, different function syntaxes for `mod`, `substr`, etc.). This allows VerdictDB to easily support new databases.[2] To add support for a new DBMS, the only part that needs to be added to VerdictDB is a thin driver that extends that DBMS's JDBC/ODBC driver and understands its SQL dialect. VerdictDB's implementation is 57K lines of code (LOC), while adding a driver for Impala, Spark SQL, and Redshift required only 55, 167, and 360 LOC, respectively.

Once the rewritten query is executed by the underlying database, Answer Rewriter adjusts the results (e.g., output format, error reporting format, confidence levels, etc.) and returns an approximate answer (and error estimates, when requested) to the original query.

2.2 Supported Queries

VerdictDB speeds up analytic SQL queries that use common aggregate functions. When VerdictDB can speed up a query, we say VerdictDB *supports* that query. Other queries are simply passed down to the underlying database unchanged, i.e., unsupported queries do not observe any speedup. Currently, VerdictDB supports queries with *mean-like* statistics, including common aggregate functions (e.g., `count`, `sum`, `avg`, `quantile`, `var`, `stddev`), and user-defined aggregates (as long as they converge to a non-degenerate distribution [61]). VerdictDB supports `count-distinct` using a function that partitions a domain into subdomains with equal cardinalities [23]. VerdictDB does not approximate extreme statistics (i.e., *min* and *max*). Although there is theoretical work on estimating extreme statistics [66], the error bounds tend to be quite large in practice. However, if a query includes both extreme statistics and other mean-like statistics, VerdictDB automatically decomposes the query into one part with extreme statistics and the other part with mean-like statistics; then, it approximately computes only the part with mean-like statistics.

VerdictDB also supports equi-joins, comparison subqueries (e.g., `where sales < (select avg(sales) ...)`), and other selection predicates (e.g., `IN list`, `LIKE regex`, `<`, `>`, and so on). When there is a comparison subquery, VerdictDB converts it into a join. For instance, consider the following query with a correlated subquery:

```
select ...
from orders t1 inner join order_products t2
  on t1.order_id = t2.order_id
where price > (select avg(price)
                 from order_products
                where product = t1.product);
```

This query produces the same results as the following query, which instead uses a join with a derived table.

```
select ...
from orders t1 inner join order_products t2
  on t1.order_id = t2.order_id
  inner join (select product, avg(price) avg_price
              from order_products
              group by product) t3
  on t2.product = t3.product
where t2.price > avg_price;
```

The above query flattening is performed for comparison subqueries. Currently, VerdictDB does not approximate other types of subqueries, e.g., `IN (select ...)`, `EXISTS (select ...)`, or subqueries in the select clause. Table 1 summarizes the types of queries supported by VerdictDB.

2.3 Workflow

The user's interaction with VerdictDB consists of two stages: sample preparation and query processing. The sample preparation stage is an *offline* process, during which the user informs VerdictDB of the tables for which AQP is desired. By default, VerdictDB automatically builds different types of sample tables based on the column cardinalities (see [58]); however, the user can also manually specify which types of sample tables to build. In general, VerdictDB may construct multiple (sizes and types of) samples for the same table. Section 3 describes the different types of sample tables that VerdictDB constructs. The metadata about the created sample tables (e.g., names, types, sampling ratios) are recorded in a specific schema inside the database catalog.

At runtime, when the user issues a query, VerdictDB first identifies the set of sample tables that can be used in place of each of the base tables that appear in the query. Then, VerdictDB's sample planning module determines a combination of sample tables that can minimize the overall approximation error given a specified I/O budget (e.g., 2% of the original data). Depending on the available samples, the I/O budget, or the query type, the sampling module may simply resort to using the base tables themselves [58].

Once a combination of sample tables is chosen to use for query processing, VerdictDB rewrites the original query into another SQL statement that, when executed by the underlying database, can simultaneously produce both an unbiased approximate answer and probabilistic error bounds. When the underlying database returns the result for the rewritten query, VerdictDB extracts and scales

[2] VerdictDB's current release comes with drivers for Apache Hive, Apache Impala, Apache Spark SQL (1.6 and 2.0), and Amazon Redshift. We plan to add drivers for Oracle, Presto, and HP Vertica in the near future.

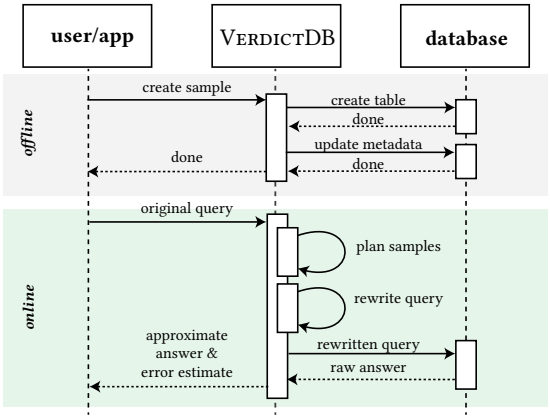


Figure 2: The offline and online workflow of VerdictDB: sample preparation (gray) and query processing (green).

the approximate answer and the error estimates, and returns to the user. This workflow is visualized in Figure 2.

For simplicity, we present our techniques assuming that the data in the original tables are static. VerdictDB can also efficiently support periodic data ingestion [58].

2.4 User Interface

Traditionally, AQP engines have allowed users to either specify a latency requirement (e.g., return an approximate answer within 2 seconds) [11, 12], or an accuracy requirement (e.g., return an answer that is at least 99% accurate) [11, 12, 33, 51]. The problem with offering a latency knob is that predicting the latency of a query in advance, even when the input size is known, is still an unsolved problem for databases [38, 44, 45].[3] For example, previous engines offering latency knobs have resorted to simple heuristics (e.g., linear regression in BlinkDB [11]), which often miscalculate actual runtimes. Likewise, predicting the approximation error before running the query is practically impossible [47]. Even when closed-form error estimation is applicable [33], the estimate depends on several query-dependent factors, such as the query selectivity, the variance of the attribute values satisfying the selection predicates, the inter-tuple correlations (when joining sample tables), etc. These query-dependent factors are hard to predict, and are typically known only after the AQP engine has run the query.

For these reasons, VerdictDB offers a more practical knob to the user, which is easier to enforce. Instead of specifying a latency or accuracy requirement, VerdictDB's users specify an I/O budget. For every table that exceeds a certain size (10 million rows, by default), users can choose a maximum percentage of the table that can be used when that table appears in analytical queries (2%, by default). Optionally, users can also specify a minimum accuracy requirement. However, VerdictDB interprets this accuracy requirement only after the query is executed and the approximation errors are estimated: if the error(s) violate the accuracy requirement, VerdictDB reruns the query on the base tables themselves and returns an exact answer back to the user. In such cases, VerdictDB uses the notion of High-level Accuracy Contract (HAC) [47], which is also adopted by SnappyData [48, 63]. Similar to previous AQP engines [8, 9, 16, 18, 34, 36, 53, 54, 60], the error semantics in VerdictDB are based on the notion of confidence intervals. For instance, 99% accuracy at 95% confidence would mean that the true answer lies between ±1% of the approximate answer with 95% probability.

Similar to Oracle 12c [67], the approximation settings in VerdictDB (e.g., I/O budget, accuracy requirements) can be set either on a per-query basis or at the system/connection level. The latter allows VerdictDB to be used in a *transparent* mode for speeding up (legacy) applications that are not designed to use AQP features, e.g., the DBA can choose appropriate settings on behalf of the application. VerdictDB does not include error estimates as additional columns in the output, unless requested by the user. Again, this is to ensure that legacy applications can seamlessly consume approximate results without the need to be modified.

3 SAMPLE PREPARATION

In Section 3.1, we briefly review the four types of sample tables used by VerdictDB: uniform samples, hashed samples, stratified samples, and irregular samples.

With the exception of stratified samples, the rest of the sample types can be constructed using SQL in a straightforward manner. For stratified samples, the sampling ratios differ for each stratum; however, adjusting the sampling probabilities dynamically while scanning the data using procedural SQL (e.g., Transact-SQL) is not applicable to all databases, and can also be much slower than standard select statements. In Section 3.2, we introduce VerdictDB's probabilistic approach to efficient construction of stratified samples, which relies only on standard select statements.

3.1 Background: Sample Types

A sample table T_s is a subset of tuples from an original table T. The inclusion probabilities (a.k.a. sampling probabilities) may differ for each tuple. In addition to the tuples themselves, VerdictDB also records their sampling probabilities as an extra column in the sample table. We can define different sample types using a real-valued sampling parameter $\tau \in [0, 1]$, a column set C (e.g., $C = \langle \texttt{city}, \texttt{age} \rangle$), and the number of unique values d_C under C.

1. **Uniform sample.** A sample T_s is uniform if every tuple in T is sampled independently (i.e, a Bernoulli process) with a sampling probability equal to τ.
2. **Hashed sample.**[4] Given a column set C, a hashed sample on C is defined as $T_s = \{t \in T \mid h(t.C) < \tau\}$, where $h(\cdot)$ is a uniform hash function that maps every value of C into a real number in $[0, 1]$, and $t.C$ is the value of C in t. Here, the sampling probabilities are all set to $|T_s|/|T|$.
3. **Stratified sample.** Given a column set C with unique values $\{c_1, \ldots, c_{d_C}\}$, a stratified sample on C is a sample that satisfies this condition:

$$\forall i = 1, \ldots, d_C : \quad \left|\sigma_{C=c_i}(S)\right| \geq \min\left(\frac{|T| \cdot \tau}{d}, \left|\sigma_{C=c_i}(T)\right|\right) \quad (1)$$

[3] It is well-known that the cost estimates provided by query optimizers are not an accurate predictor of actual latencies.

[4] Hashed samples are also called *universe samples* [28, 33].

The sampling probability of a tuple with c_i in C is set as $\frac{|\sigma_{C=c_i}(S)|}{|\sigma_{C=c_i}(T)|}$.

4. **Irregular sample.** When the sampling probabilities do not meet any of the properties mentioned above, we call it an irregular sample.

During the sample preparation stage, VerdictDB only constructs sample tables that belong to one of the first three types, i.e., uniform sample, hashed sample, and stratified sample. Irregular samples may appear only during query processing as a result of joining other (sample) tables. By default, VerdictDB uses 1% for τ so that the sample sizes are within the default query-time I/O budget (i.e., 2%).

3.2 Probabilistic Stratified Samples

This section presents VerdictDB's SQL-based, easily parallelizable approach to constructing stratified samples. VerdictDB takes a two-pass approach for creating stratified samples: in the first pass, the group sizes are computed; in the second pass, tuples are sampled according to group-size-dependent sampling probabilities, as follows.

```
select *
from orders inner join T_temp
     on T.c1 = T_temp.c1 and ... and
        T.ck = T_temp.ck
where rand() <
  (sampling_prob_expression)
```

where `T_temp` is the table constructed in the first pass with the schema ⟨`c1`, ..., `ck`, `strata_size`⟩. Here, ⟨`c1`, ..., `ck`⟩ is the column set of the stratified sample, and `(sampling_prob_expression)` is a SQL expression that determines the sampling probability for each tuple, which we describe in detail below.

Note that the tuples here are sampled independently from one another (i.e., Bernoulli process). The key advantages of this approach are that (1) the sampling process can easily be expressed in SQL, and (2) its operations can be executed in parallel.

However, the downside here is that the guarantee in Equation 1 may no longer hold. This is because a Bernoulli process does not produce a sample with exactly p% of the tuples. For example, suppose that we need to sample at least 10 tuples out of a stratum of 100 tuples (i.e. `strata_size` = 100). If we use a Bernoulli process with a sampling ratio of 0.1 (10 / 100), we will have fewer than 10 tuples with probability $\sum_{k=0}^{9} \binom{100}{k} 0.1^k \, 0.9^{100-k} \approx 0.45$. In other words, a naïve approach would violate the guarantee of Equation 1 for nearly half of the strata.

To guarantee Equation 1, VerdictDB uses a staircase function by substituting `(sampling _prob_expression)` with a case expression, i.e., `(case strata_size > 2000 then 0.01 when strata_size > 1900 then 0.012 ... else 1)`. The staircase function expressed in a case expression upper-bounds $f_m(n)$, where $f_m(n)$ is a value such that a Bernoulli process with ratio $f_m(n)$ samples at least m out of n tuples with probability $1-\delta$ (by default, δ=0.001). VerdictDB uses the following lemma to determine $f_m(n)$ (proof deferred to Appendix C).

Lemma 1. *Let a sample be constructed by Bernoulli sampling from n tuples with p sampling probability. Then, the sampling probability for outputting at least m tuples with probability $1-\delta$ is*

$$f_m(n) = g^{-1}(m; n)$$

$$\text{where} \quad g(p; n) = \sqrt{2n \cdot p(1-p)}\ \text{erfc}^{-1}\left(2(1-\delta)\right) + np$$

erfc^{-1} is the inverse of the (standard) complementary error function.

4 VARIATIONAL SUBSAMPLING: PRINCIPLE

In this section, we describe VerdictDB's novel error estimation technique. Previous AQP engines, especially those that support general analytical queries, have relied on bootstrap [73], which belongs to a family of error estimation techniques called *resampling* [15, 35]. Resampling techniques, despite various optimizations [10, 60], are still too expensive to be implemented at a middleware layer. Therefore, we propose the use of a different class of error estimation techniques, called *subsampling*, for the first time in an AQP context. Although subsampling is, in general, much more efficient than resampling, direct application of subsampling theory can be still quite daunting.

In the remainder of this section, we first provide a general overview of subsampling (Section 4.1), and explain why its traditional variant is too expensive. We then propose a new variant, called *variational subsampling*, which dramatically reduces the cost of traditional subsampling without compromising its statistical correctness (Section 4.2). Later, in Section 5, we generalize this idea to more complex queries, such as nested queries and joins.

4.1 Subsampling Basics

Before presenting the basics of subsampling theory, we first discuss bootstrap. Bootstrap is the state-of-the-art error estimation mechanism used by previous AQP engines, especially those that support general analytical queries [7, 16, 29, 34, 60, 73]. We then discuss why subsampling is more amenable to efficient execution than bootstrap.

Bootstrap — Let $g(\cdot)$ be an aggregate function (e.g., mean, sum), which we wish to compute on N real values $x_1, \ldots, x_N$ (e.g., values of a particular column), i.e., $g(x_1, \ldots, x_N)$. Let a simple random sample of these N values be $X_1, \ldots, X_n$, and $\hat{g}(\cdot)$ be an estimator of $g(\cdot)$.[5] That is, we can estimate $g(x_1, \ldots, x_N)$ using $\hat{g}_0 = \hat{g}(X_1, \ldots, X_n)$. In an AQP context, we also need to measure the quality (i.e., expected error) of the estimate $\hat{g}(X_1, \ldots, X_n)$.

To measure the quality of the estimate, bootstrap recomputes the aggregate on many resamples, where each resample is a simple random sample (with replacement) of the original sample. In bootstrap, the size of a resample is the same as the sample itself, i.e., some of the elements $X_1, \ldots, X_n$ might be missing and some might be repeated, but the total number remains as n. Let $\hat{g}_j$ be the value of the estimator computed on the j-th resample, and b the number of resamples (b is usually a large number, e.g., 100 or 1000). Bootstrap uses $\hat{g}_1, \ldots, \hat{g}_b$ to construct an empirical distribution of the sample statistics, which can then be used to compute a confidence interval. Let $\hat{g}_0$ be the estimator's value on the original sample itself, and t_α be the α-quantile of $\hat{g}_0 - \hat{g}_j$. Then, the $1-\alpha$ confidence interval can

[5] For example, $\hat{g}(\cdot)$=$g(\cdot)$ when g is avg, but $\hat{g}(\cdot)$=$\frac{N}{n} g(\cdot)$ when g is sum.

be computed as:

$$\left[\hat{g}_0 - t_{1-\alpha/2} \quad , \quad \hat{g}_0 - t_{\alpha/2}\right]$$

Due to its generality, bootstrap has been used in many different domains [49]. Although there is an I/O-efficient variant of bootstrap, called *consolidated bootstrap* [10], its computational overhead remains high, due to the repetitive computation of the aggregate, which has a time complexity of $O(n \cdot b)$. Another variant, called *analytical bootstrap* [73], reduces the computational cost but requires modifying the relational operators inside the database (thus, inapplicable to VerdictDB, which is a middleware).

Subsampling — Subsampling follows a procedure similar to bootstrap, but with two key differences: (1) instead of resamples, it uses *subsamples* which are much smaller, and (2) instead of drawing tuples from the original sample with replacement, subsampling draws tuples without replacement. In other words, a subsample is also a simple random sample of the original sample, but without replacement, and of size n_s where $n_s \ll n$. In general, n_s must be chosen such that it satisfies the following two conditions [61]: (1) $n_s \to \infty$ as $n \to \infty$, and (2) $n_s/n \to 0$ as $n \to \infty$. Once the subsamples are constructed, the time complexity of the aggregation is only $O(n)$; however, constructing the subsamples can itself take $O(b \cdot n)$. We discuss this in more detail shortly.

Computing the $1-\alpha$ confidence interval is similar to bootstrap, but requires a scaling:

$$\left[\hat{g}_0 - t_{1-\alpha/2} \cdot \sqrt{n_s/n} \quad , \quad \hat{g}_0 - t_{\alpha/2} \cdot \sqrt{n_s/n}\right]$$

In theory, the difference between the empirical confidence interval and the true interval is $O(b^{-1/2} + b/n)$ [61].

While more efficient than bootstrap (since $n_s \ll n$), performing subsampling as a middleware can still be quite expensive. We illustrate this inefficiency by exploring a few possible implementations of subsampling in SQL, which is what a middleware would have to do. (We empirically compare various error estimation techniques in Section 6.4.)

Implementing Subsampling in SQL — Suppose we need to compute `sum(price)` of the `orders` table grouped by `city`. Also, let `orders_sample` be a sample table of the `orders` table. One can implement subsampling both with and without User Defined Aggregates (UDAs). As a toy example, suppose n=1M, n_s = 10K, b = 100.

To implement traditional subsampling without UDAs, one needs to first construct a temporary table, say `orders_subsamples` which, in addition to the original columns, must also have an additional column, say `sid`, indicating the subsample that each tuple belongs to. Here, the `sid` column would contain the subsample id (integers between 1 and b). Given that some tuples might belong to multiple subsamples, the same tuple may appear multiple times, but each time with a different `sid`.[6] However, there should be exactly n_s tuples with the same `sid`. Given such a table, one can use the following query to compute the aggregate on b=100 different subsamples (scaling factors omitted for simplicity):

```
select city,
  sum(price * (case when sid = 1 then 1 else 0)),
      ...
  sum(price * (case when sid = b then 1 else 0))
from orders_subsamples
group by city;
```

Query 1: Performing traditional subsampling without UDAs.

This query costs $O(b \cdot n_s)$, but constructing the `orders_subsamples` table itself costs $O(b \cdot n)$.[7]

When the underlying database supports UDAs, one can avoid the need for constructing the `orders_subsamples` table, and produce the subsamples and their aggregates in a single scan. Let `subsum(price)` be a UDA that, while making a single pass on the original sample, maintains a random subset of exactly n_s tuples using reservoir sampling, and at the end of the scan returns the sum of the `price` values for the selected tuples. Then, one can use the following query, whereby 100 instances of the UDA will each return the aggregate value on a separate subsample:

```
select city,
       subsum(price) as subsample_agg1,
       ...
       subsum(price) as subsample_agg100
from orders_sample
group by city;
```

Query 2: Performing traditional subsampling using UDAs.

Assuming an ideal case, where the underlying database uses a shared scan among all the UDAs, the time complexity is still $O(b \cdot n)$, i.e., b UDAs each reading n tuples.

Next, we propose a novel variant of subsampling, which we call *variational subsampling*. We show that our variant has a time complexity of $O(n)$, and is hence much more efficient than traditional subsampling.

4.2 Variational Subsampling

In this section, we introduce our new subsampling technique, called *variational subsampling*, which relaxes some of the requirements of traditional subsampling. We show that our proposal, while significantly more efficient, still retains the statistical correctness of traditional subsampling. In the following subsections, we will generalize our idea to more complex queries.

Core Idea — In traditional subsampling, the same tuple *must* be able to belong to multiple subsamples, and each subsample *must* be exactly of size n_s. Enforcing these restrictions is a major source of computational inefficiency. Our proposed technique relaxes these restrictions, by (1) allowing each tuple to belong to, at most, one subsample, and (2) allowing the sizes of different subsamples to differ. Surprisingly, our analysis reveals that the asymptotic properties of subsampling continue to hold despite lifting these restrictions. The only caveat is that one must scale the aggregates accordingly (Theorem 2). However, these two relaxations make a critical difference in terms of computational efficiency: for each tuple, we now only need to generate a single random number to determine which subsample it belongs to (if any), and then perform the aggregation only once per tuple, instead of repeating this process b times.

To state this process more formally, we first need to define a *variational table*. We then explain how to populate this table efficiently.

[6] Here, we could just keep the aggregation column instead of the entire tuple.

[7] Note that these subsamples should not be precomputed offline and reused for every query, due to the risk of *consistently incorrect estimates* [40].

Definition 1. ***(Variational Table)*** *Let b be the desired number of subsamples. A variational table is a sample table augmented with an extra column that is populated by random integers between 0 and b (inclusive), generated independently according to the following weights: $(n-b\cdot n_s,\ n_s,\ n_s,\ \ldots,\ n_s)$. That is, 0 is chosen with probability $\frac{n-b\cdot n_s}{(n-b\cdot n_s)+b\cdot n_s} = \frac{n-b\cdot n_s}{n}$ and each of the integers $1, \cdots, b$ are chosen with probability $\frac{n_s}{(n-bn_s)+b\cdot n_s} = \frac{n_s}{n}$. An integer between 1 and b indicates the subsample id that the current tuple belongs to, whereas 0 indicates that the tuple does not belong to any subsamples.*

Note that the independent sampling is an *embarrassingly parallel* process, but it also means that the subsamples are no longer guaranteed to be of size n_s. Later, we show how, with proper scaling, we can still achieve the same asymptotic properties as those offered by traditional subsampling.

Nonetheless, a variational table can be populated in a straightforward fashion. As we scan the sample table, we randomly assign a single `sid` (i.e., subsample id) to each tuple. An `sid` between 1 and b indicates that the tuple belongs to the subsample represented by that integer, while an `sid` of 0 indicates that the tuple does not belong to any subsample. Since each tuple is assigned to one subsample at most, this approach effectively partitions the sample into b subsamples, plus the set of those tuples that are not used in any subsample. The tuples belonging to different subsamples can be aggregated separately in SQL using a `group-by` clause.

Query 3 illustrates how this process can be expressed in a single SQL statement, using a toy example, with n = 10M, n_s = 10K, and b = 100:

```
select *, 1+floor(rand() * 100) as sid
from orders_sample
where 1+floor(rand() * 1000) <= 100;
```

Query 3: Example of creating a variational table.

This query randomly assigns, on average, n_s tuples to each of the b non-intersecting subsamples. To implement the weighted sampling, it uses the expression `1+floor(rand() * 1000)`, which returns a random integer between 1 and 1000 (inclusive) with equal probability. Values outside the range [1, 100] are treated as 0, and are discarded accordingly. This is because a tuple should not belong to any of the subsamples, with probability $\frac{n-b\cdot n_s}{n} = \frac{10M-100*10K}{10M} = 0.9$. Once such tuples are discarded, the remaining tuples have an integer in the range $[1, b]$, representing their `sid`. Note that even if the two instances of the `rand()` function (in the `select` and `where` clauses) return different values for the same tuple, the overall probabilities remain the same.

Query 3 generates the variational table with $O(n)$ operations, and can be embedded in another query to perform the aggregation on its output. Below is an example of how to perform the entire variational subsampling in a single query.

```
select city, sum(price), count(*) as ns
from (select *, 1+floor(rand() * 100) as sid
       from orders_sample
       where 1+floor(rand() * 1000) <= 100
     ) as orders_v
group by city, sid;
```

Query 4: Example of variational subsampling.

Note that here we are also returning the size of each subsample (the `ns` column). This is because, unlike traditional subsampling, our subsamples might vary in size; as we discuss in Theorem 2, variational subsampling uses these sizes to correct its distribution of the sample estimate.

Nonetheless, it is easy to see that Query 4 is considerably more efficient than traditional subsampling. Query 4 performs two aggregates per each of the $b \cdot n_s$ tuples in the `orders_v`; thus, the aggregation cost is $O(b \cdot n_s)$. Since the cost of the inner query (building the `orders_v`) is $O(n)$ and $b \cdot n_s \ll n$, the overall time complexity of Query 4 is only $O(n + b \cdot n_s)$=$O(n)$. Therefore, variational subsampling is at least $O(b)$ times more efficient than traditional subsampling, which costs $O(b \cdot n)$ operations (see Section 4.1).

Error Correction — To guarantee that the distribution of the subsample aggregates converges to the true distribution of the aggregate on the original sample, variational subsampling has to correct for the varying sizes of its subsamples. Let $n_{s,i}$ denote the size of the i-th subsample. Below, we formally show that the following empirical distribution converges to the true distribution of a sample estimate:

$$L_n(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\sqrt{n_{s,i}}(\hat{g}_i - \hat{g}_0) \le x\right) \tag{2}$$

where x is the deviation of a subsample aggregate from the sample aggregate, and $\mathbb{1}()$ returns 1 if its argument is true; and 0 otherwise. (Recall that $\hat{g}_0$ and $\hat{g}_j$ are the values of the estimator computed on the original sample and the i-th resample, respectively.)

Theorem 2. *Let $J_n(x)$ denote the (non-degenerate) true distribution (cumulative distribution function) of the estimate based on a sample of size n. Then, for any n_s such that $n_s \to \infty$ and $n_s/n \to 0$ as $n \to \infty$,*

$$L_n(x) \to J_n(x)$$

in distribution as $n \to \infty$.

This theorem implies that, when n is large, variational subsampling can correctly estimate the distribution of a sample estimate. The proof to this theorem is presented in Appendix C. In Appendix B.3, we show that variational subsampling's asymptotic error is minimized when one chooses $n_s = n^{1/2}$. This is why VerdictDB uses $n_s = n^{1/2}$ as default, but users can choose different values. In Section 6.5 and Appendix B.3, we compare the error of bootstrap, traditional subsampling, and variational subsampling.

In this section, we used a simple query to illustrate variational subsampling. Next, we show how to obtain a variational table for more complex queries.

5 VARIATIONAL SUBSAMPLING ADVANCED

According to Theorem 2, as long as we can construct a variational table for a query, we can correctly estimate the distribution of its sample estimate. In this section, we extend our core idea from Section 4 to obtain variational tables for joins (Section 5.1) and nested subqueries (Section 5.2).

5.1 Variational Subsampling for Joins

Handling joins is a challenging task for all AQP solutions due to two main problems. The first problem is joining (uniform) samples leads to significantly fewer tuples in the output [18]. The second

problem is that joining sampled tables leads to inter-tuple dependence in the output [39]. To address the first problem, existing AQP solutions[8] use at most one sampled relation per join [8], or require the join key to be included in the stratified sample [11] or a hashed sample [33]. VerdictDB uses the same strategies for sidestepping the low cardinality of the join, and focuses on solving the second problem. This is because even when the first problem can be solved by the aforementioned solutions, efficient accounting of the inter-tuple correlations is still a challenge.

To address the second problem, previous solutions have either made strong foreign-key (FK) assumptions on the join key [8], or have used Horvitz-Thompson (HT) estimators [33] and resampling techniques [60] to account for inter-tuple correlations. As a middleware, VerdictDB cannot enforce FK relationships, and expressing HT estimators for correlations [33] in SQL will involve expensive self-joins. Also, as mentioned earlier, resampling strategies are too costly for a middleware. Instead, VerdictDB extends its variational subsampling to automatically account for inter-tuple correlations, in a manner that can easily be expressed in SQL and efficiently executed by the underlying database. From a high-level, to use variational subsampling for a join, we need to construct a variational table of the join output. In the rest of this section, we explain how to efficiently obtain a variational table of a join.

Suppose a query involves an aggregation over the join of the `orders` and `products` tables, and VerdictDB decides to use their respective sample tables to compute an approximate output. To estimate the quality of this approximate answer, VerdictDB uses the variational tables of the source relations, i.e., `orders` ⋈ `products`.

A basic approach to constructing a variational table of the join is as follows. Given the variational tables of the two tables, i.e., `orders_v` and `products_v`, join each subsample of the first table with its corresponding subsample from the other table to construct a new subsample, which we call a *joined subsample*. Repeat this process b times to construct a variational table of the join query. The following theorem guarantees the correctness of this approach (see Appendix C for proof).

Theorem 3. *Let $g(T,S)$ be an aggregate function involving two tables T and S, and $\hat{g}(T_s,S_s)$ be an estimator of $g(T,S)$, where T_s and S_s are respective samples of T and S. Furthermore, let $T_{s,i}$ and $S_{s,i}$ be the i-th subsamples of T_s and S_s, respectively. Lastly, let $n_{s,i}$ denote the size of the join of $T_{s,i}$ and $S_{s,i}$. If $|T_s|/|T_{s,i}| = |S_s|/|S_{s,i}|$, then $L(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\sqrt{n_{s,i}}\,(\hat{g}(T_{s,i},S_{s,i}) - \hat{g}(T_s,S_s)) \le x\right)$ converges to the true distribution of $\hat{g}(T_s,S_s)$ as $n \to \infty$.*

In other words, the above theorem states that we can estimate the distribution of our sample-based join approximation, $\hat{g}(T_s,S_s)$, by recomputing the join on respective subsamples of T_s and S_s, namely T_{s_i} and S_{s_i}. However, implementing this approach in SQL would entail a `union` of multiple join expressions, resulting in an extremely inefficient query plan.

In VerdictDB, we take a significantly more efficient approach, based on a key observation. We formally show that, instead of repeatedly joining multiple subsamples, it suffices to simply join the two variational tables only once, followed by reassigning the their `sid` values using a special function (formally introduced in Equation 4). We show that this approach requires only a single join and a single projection; thus, it can easily (and efficiently) be implemented in SQL. To prove the correctness of this approach, we first need to explain the basic approach more formally.

[8]Others have resorted to online sampling of the joined relations [27, 39]; however, as a middleware, VerdictDB is currently based on offline sampling.

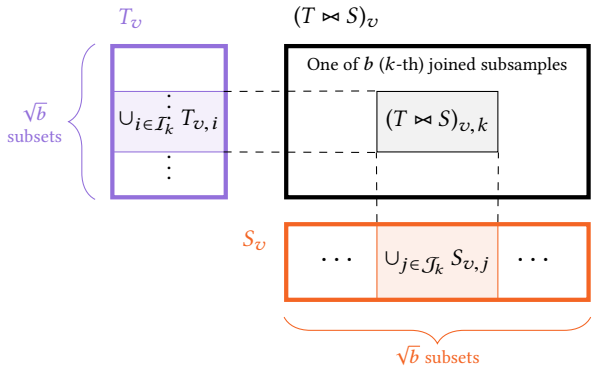


Figure 3: Joining variational tables to construct a new variational table of a join. Instead of repeatedly joining $\sqrt{b}\times\sqrt{b}$ pairs of subsamples, we simply join the two variational tables (only once), then reassign their `sid` values.

Basic Approach — To produce a single joined subsample of `orders` ⋈ `products`, we need to join $\sqrt{b}$ subsamples of `orders_v` with $\sqrt{b}$ subsamples of `products_v` to produce a joined subsample, where b is the number of the subsamples in each table.

Before a formal presentation, we first use a toy example. Suppose `orders_v` and `products_v` each contain b=100 subsamples of size 10K. Suppose their join, namely `orders_v` ⋈ `products_v`, has 1M tuples. Observe that the probability of two randomly chosen tuples from `orders_v` and `products_v` satisfying the join condition is $\frac{1M\cdot 1M}{1M} = \frac{1}{1M}$. We can calculate the number of subsamples needed from each of `orders_v` and `products_v` to yield a joined subsample of size 1M/100 = 10K. Let this number be x. Since the join probability is 1/1M, x must satisfy $\frac{(x\times 10K)\cdot(x\times 10K)}{1M} = 10K$. This means x=10. In this example, we see that $\sqrt{b} = \sqrt{100}$=10=x.

To formally express this process using relational algebra, let T_v and S_v denote the variational tables of two original tables T and S, respectively. Further, denote the i-th and j-th subsamples in those variational tables by $T_{v,i}$ and $S_{v,j}$. Also, let $\mathcal{I}$ and $\mathcal{J}$ be index sets with integers from 1 to b, i.e., $\mathcal{I}=\mathcal{J}=\{1,2,\ldots,b\}$. Then,

$$(T \bowtie S)_{v,k} \;=\; \cup_{i\in\mathcal{I}_k} T_{v,i} \;\bowtie\; \cup_{j\in\mathcal{J}_k} S_{v,j} \tag{3}$$

where $\mathcal{I}_k$ is a subset of $\mathcal{I}$, $\mathcal{J}_k$ is a subset of $\mathcal{J}$, and each of $\mathcal{I}_k$ and $\mathcal{J}_k$ includes $\sqrt{b}$ elements. Since $(T \bowtie S)_v = \cup_{k=1,\ldots,b} (T \bowtie S)_{v,k}$, the join operation on the right-hand side of Equation 3 must be repeated b times.

Efficient Approach — Our key observation is that the variational table of the join, or equivalently, the union of $(T \bowtie S)_{v,k}$ for $k = 1,\ldots,b$, can be converted into a logically equivalent but computationally more efficient expression, if the cross product of $\mathcal{I}_k$ and $\mathcal{J}_k$ partitions $\mathcal{I}\times\mathcal{J}$; that is,

$$\mathcal{I}\times\mathcal{J} \;=\; \bigcup_k (\mathcal{I}\times\mathcal{J})_k \;=\; \bigcup_k \mathcal{I}_k\times\mathcal{J}_k$$

We state our result formally in the following theorem (proof deferred to Appendix C).

Theorem 4. *If there exists*

$$h(i,j) = k \quad \textit{for } (i,j) \in (\mathcal{I} \times \mathcal{J})_k = \mathcal{I}_k \times \mathcal{J}_k \tag{4}$$

then

$$(T \bowtie S)_v = \Pi_{*,\, h(i,j) \text{ as sid}} \; (T_v \bowtie S_v) \tag{5}$$

In this theorem, the projection does not remove duplicates, and the $*$ subscript in the projection means that we preserve all of the existing columns. The final projection with "$h(i,j)$ as `sid`" effectively identifies the subsamples in $(T \bowtie S)_v$, namely, the variational table for $T \bowtie S$. Note that, given such an $h(i,j)$ function, the expression in Equation 5 can easily be expressed in SQL.

We give an example of the function $h(i,j)$ in Equation 4:

$$h(i,j) = \left\lfloor \frac{i-1}{\sqrt{b}} \right\rfloor \cdot \sqrt{b} + \left\lfloor \frac{j-1}{\sqrt{b}} \right\rfloor + 1 \qquad i,j = 1,\ldots,b$$

where $\lfloor \cdot \rfloor$ returns the floor of its argument. Note that this $h(i,j)$ function is similar to how two-dimensional arrays are indexed sequentially in most programming languages (e.g., C).

Figure 3 visually explains our approach. Sets of subsamples from T_v and S_v are joined to produce the k-th joined subsample. Each set contains $\sqrt{b}$ subsamples, and there are $\sqrt{b} \cdot \sqrt{b} = b$ combinations. Thus, joining every pair of sets (of subsamples) produces b joined subsamples in total. Since the hash function $h(i,j)$ can identify k given i and j, we can simply join all tuples first, and then assign new `sid` values.

5.2 Variational Subsampling for Nested Queries

To illustrate how VerdictDB obtains a variational table for nested queries, consider the following query as an example:

```
select avg(sales) as avg_sales
from (select city, sum(price) as sales
      from orders
      group by city) as t;
```

Query 5: An aggregate query in the from clause.

For variational subsampling, we need a variational table of `t`, which we denote by `t_v`.

Note that `t_v` should be a union of b aggregate statements, where each aggregate statement is computed on a subsample. Let a variational table of `orders` be `orders_v` (which includes an `sid` column to indicate the subsample that each tuple belongs to). Then, a basic approach to obtaining `t_v` is

```
select city, sum(price) as sales, avg(1) as sid
from orders_v
where sid = 1
group by city
union
...
union
select city, sum(price) as sales, avg(b) as sid
from orders_v
where sid = b
group by city;
```

Query 6: A basic approach to obtaining a variational table of `t`.

However, by exploiting the property that the subsamples in `orders_v` are disjoint, we can perform the above operations more efficiently. Formally, let T_v be a variational table. Then,

$$\begin{aligned} \bigcup_k {}_G\mathcal{G}_g \, (T_{v,k}) &= \bigcup_k {}_G\mathcal{G}_g \; (\sigma_{\texttt{sid}=k}(T_v)) \\ &= {}_{G,\texttt{sid}}\mathcal{G}_g \, (T_v) \end{aligned} \tag{6}$$

where ${}_G\mathcal{G}_g$ is an aggregate operator with a set of grouping attributes G and an aggregate function g. Also, $T_{v,k}$ is the k-th subsample of T_v, as defined in Section 5.1.

Equation 6 indicates that Query 6, i.e., the variational table of `t`, can be alternatively expressed using the variational table of `orders`, i.e., `orders_v`:

```
select city, sum(price) as sales, sid
from orders_v
group by city, sid;
```

Query 7: A variational table of an aggregate statement.

Finally, Query 7 can be used in place of `t` in Query 5 for estimating the quality of a sample estimate. Note that Query 7 requires $O(b)$ fewer scans (of `orders_v`) than Query 6.

6 EXPERIMENTS

In this section, we empirically evaluate VerdictDB. Our experiments aim to demonstrate VerdictDB's platform-independence, efficiency, and statistical correctness. In summary, our experiments show the following:

1. Thanks to its UAQP, VerdictDB delivered an average of 18.45×—and up to 171×—speedup (for Impala, Spark SQL, and Redshift), and with less than 2.6% relative error. (Section 6.2, Appendix B.1)
2. VerdictDB's performance was comparable to (and sometimes even faster than) a tightly-integrated, commercial AQP engine, i.e, SnappyData.[9] (Section 6.3)
3. Variational subsampling was 348× faster than traditional subsampling and 239× faster than consolidated bootstrap [10] expressed in SQL. (Section 6.4)
4. Variational subsampling yielded statistically correct estimates. (Section 6.5)

For interested readers, Appendix B.2 offers additional experiments on VerdictDB's offline sample preparation overhead.

6.1 Setup

SQL Engines and Clusters — We used Spark 1.6.0 and Impala 2.8.0 included in CDH 5.11.2. For Spark SQL and Impala experiments, we used 10 EC2 `r4.xlarge` instances as workers and another one as a master. Each instance had Intel Xeon E5-2686 v4 processors (4 cores), 30.5 GB memory, and 500 GB SSD for HDFS. For Redshift experiments, we used 20 `dc1.large` instances as workers and an additional one as a master. Each instance had a CPU with 2 cores, 15 GB memory, and 160 GB SSD.

Datasets and Queries — We used three datasets:

1. `insta` [3]: This is a 100× scaled sales database of an actual online grocery store called Instacart. The size of the dataset was 124 GB before compression.

[9] In our experiments, we used SnappyData's community edition version 0.8 (SnappyData's more recent versions are likely to perform better than this version).

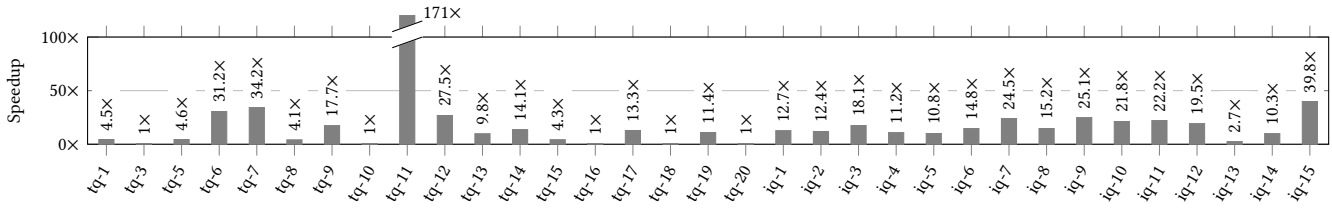


Figure 4: VerdictDB's speedups for Impala. Associated errors are in Figure 10. (Spark and Redshift deferred to Figure 9.)

2. `TPC-H` [6]: This is a 500 GB standard TPC-H dataset.
3. `synthetic`: This is a synthetic dataset we generated to fine-control various properties of data (defined in Section 6.5).

Spark SQL and Impala loaded and processed the Parquet-compressed data from an SSD-backed HDFS; Amazon Redshift automatically stored them in a compressed columnar format.

VerdictDB created sample tables for large fact tables: 1% uniform samples, 1% universe samples, and up to 80% budget for stratified samples. (We used a larger budget for stratified samples since the `TPC-H` dataset included many high-cardinality columns.)

We used 33 queries in total: 18 out of the 22 `TPC-H` queries[10] (numbered as tq-# where # is the `TPC-H` query number [6]) plus 15 micro-benchmark queries on the `insta` dataset (numbered as iq-1, . . ., iq-15). The micro-benchmark queries consisted of various aggregate functions on up to 4 joined tables. We used low-cardinality columns (up to 24, randomly chosen) in the grouping attributes of these micro-benchmark queries.

6.2 VerdictDB's Speedup for Various Engines

This section compares the query latencies of Impala, Spark SQL, and Redshift with and without VerdictDB. Since VerdictDB performs AQP, their query latencies with VerdictDB are expected to be lower. However, the purpose of this section is to (1) quantify the extent of the speedup that VerdictDB can deliver as a UAQP running on top of existing platforms, and (2) verify VerdictDB's ability in supporting common forms of OLAP queries. Moreover, testing VerdictDB with several different engines sheds light on the characteristics of a SQL engine that are favorable for UAQP.

We ran each of the 33 queries on Impala with and without VerdictDB, and measured their query latencies. We repeated the same process for Spark SQL and Redshift. Figure 4 reports VerdictDB's speedups for Impala, i.e., the latency of the regular engine divided by VerdictDB's latency. For 3 out of the 18 `TPC-H` queries (tq-3, tq-8, and tq-15), VerdictDB determined that AQP was not feasible due to the high cardinality of the grouping attributes; thus, VerdictDB simply ran the original queries (i.e., no speedup). For other queries, VerdictDB yielded 1.05×–171× speedups, with an average speedup of 18.6×. The associated errors were less than 2.6% for all queries (per-query errors are reported in Figure 10). The average speedups for Spark SQL and Redshift were 12.8× and 24.0×, respectively (their detailed results are deferred to Figure 9, due to space limitation).

Across these three engines, the speedups were larger when the default overhead of the original engine (e.g., reading the catalog) was a smaller portion of the overall query processing time. This is because VerdictDB (and AQP in general) reduces the data processing time, not the query preparation time. This depended on two factors: the default overhead, and the data preparation time. VerdictDB brought a larger speedup when the engine spent less time on catalog access and query planning (e.g., larger speedup for Spark than Redshift). Likewise, when the engine processed less prepared data, VerdictDB's speedups were more dramatic (e.g., csv file[11] versus parquet format).

[10] One query (tq-2) had no aggregates, and the other three included an `EXISTS` condition, which VerdictDB currently does not support. tq-4 and tq-20 are not supported by previous AQP engines either [11, 73].

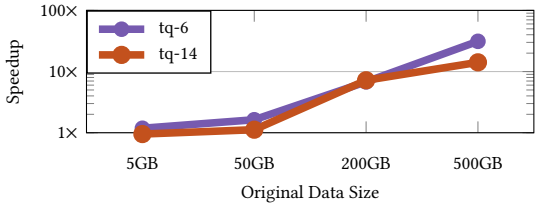


Figure 5: Speedups for different data sizes using two queries (sample fixed to 5GB; Impala).

Next, we also measured the speedups for different ratios of the sample size to the original data size. Specifically, we used a fixed sample size of 5 GB, while varying the size of the original data from 5 GB to 500 GB. Figure 5 depicts the results for two queries: tq-6 and tq-14. As expected, when the data size is already small (i.e., 50 GB), there was less room for speedup, i.e., only 1.4× on average; however, the speedup increased for larger data sizes: 7.00× for 200 GB and more than 22.6× for 500 GB.

6.3 UAQP versus Tightly Integrated AQP

This section compares the query latencies of VerdictDB, as the first example of UAQP, to tightly-integrated AQP systems. We first compare VerdictDB to a tightly-integrated sampling-based AQP engine, SnappyData; then, we compare VerdictDB to non-sampling-based (adhoc) AQP features natively offered by commercial engines (e.g., HyperLogLog implementation of `count-distinct`).

Due to its generality, middleware architecture, and sole reliance on SQL-based computations, VerdictDB is expected to be slower than tightly-integrated AQP engines that are highly specialized for a particular query engine (e.g., SnappyData). However, our goal here is to understand the extent to which VerdictDB has traded off raw performance in exchange for greater generality and deployability.

[11] When we ran the same set of queries on Impala and Spark with csv files, we observed 56.9× average speedups.

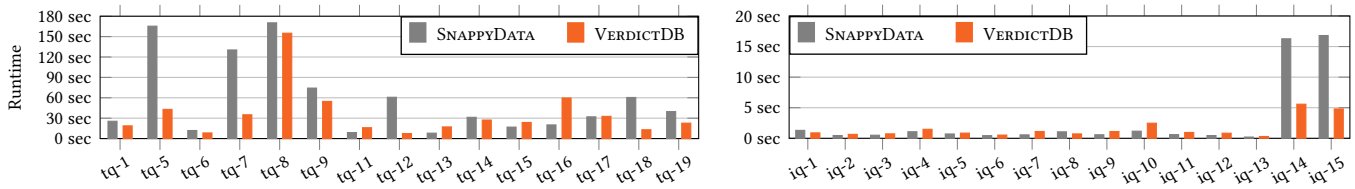


Figure 6: AQP performance of VerdictDB vs. SnappyData. VerdictDB was faster for the queries including joins of samples.

VerdictDB+Impala	Impala	VerdictDB+Redshift	Redshift
1.1 sec (0.01%)	17.1 sec (3.4%)	0.5 sec (0.02%)	7.7 sec (5.0%)

(a) approximate count-distinct runtime and relative error

VerdictDB+Impala	Impala	VerdictDB+Redshift	Redshift
1.5 sec (0%)	53.2 sec (0%)	1.0 sec (0%)	106.6 sec (0%)

(b) approximate median runtime and relative error

Table 2: Sampling-based AQP vs. native approximation.

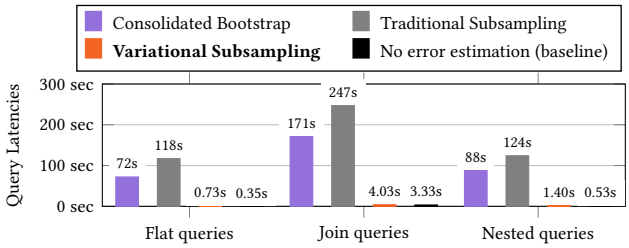


Figure 7: Runtime with different error estimation methods.

First, we compared VerdictDB on Spark SQL against SnappyData. SnappyData is tightly-integrated into Spark SQL. For these experiments, we ran the same set of TPC-H and insta queries.[12] Figure 6 reports the per-query latencies. For most queries, VerdictDB's performance was comparable to SnappyData. However, there were several queries (i.e., tq-5, tq-7, tq-12, iq-14, iq-15) for which VerdictDB was significantly faster. This is because those queries included joins of two samples. Unlike VerdictDB, SnappyData does not support the join of two samples (even when the join key is included in a stratified or hashed sample). In those situations, SnappyData simply used the original table for the second relation, while VerdictDB relied on its hashed samples.

Second, we compared VerdictDB's sampling-based approximations for count-distinct and median against Impala and Redshift's native approximate aggregates (i.e., ndv, approx_median, percentile_disc). Table 2 summarizes the results. On average, VerdictDB's sampling-based results were 43.5× faster than the native approximations. This is because Impala and Redshift's approximate aggregates rely on sketching techniques that require a full scan over data. As such, their disk I/O cost is higher.

In summary, this experiment confirms that VerdictDB's much greater generality (i.e., UAQP) comes at only a negligible loss of performance compared to tightly-integrated AQP systems.

6.4 Variational Subsampling: Efficiency

In this section, we compare the runtime overhead of three resampling-based error estimation methods: consolidated bootstrap, traditional subsampling, and variational subsampling. First, we ran three types of queries (flat, join, and nested) without any error estimation. We then ran each query with each of these three error estimation methods. By subtracting the query latencies without error estimation, we derived the runtime overhead of each error estimation technique.

Figure 7 reports the query latencies. Both consolidated bootstrap and traditional subsampling yielded substantial runtime overhead. Recall that their time complexities are $O(b \cdot n)$. In contrast, variational subsampling added only 0.38–0.87 seconds to the latency of the queries. The latency overhead comes from sample planning (26 ms on average) and extra groupby and aggregation processes inserted for performing variational subsampling. Compared to consolidated bootstrap (which is the state-of-the-art error estimation strategy [10, 71]), variational subsampling was 189×, 237×, and 100× faster, respectively. Considering the overall query latencies—including the cost of computing the approximate answers and their error bounds—running queries with variational subsampling was 99×, 42×, and 63× faster than consolidated bootstrap for flat, join, and nested queries, respectively.

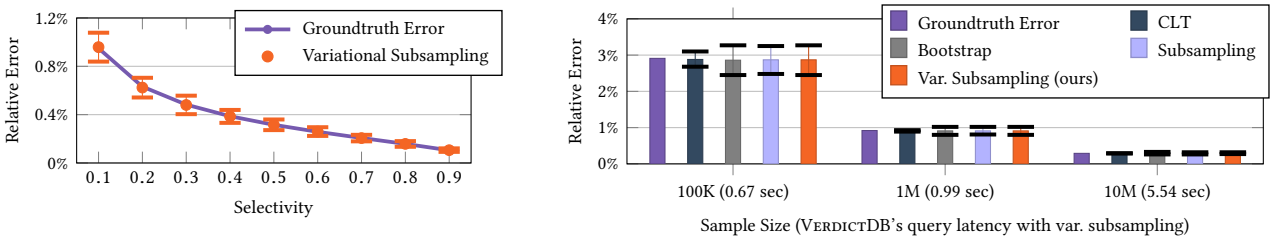
6.5 Variational Subsampling: Correctness

We study the impact of different parameters on the accuracy of variational subsampling: the query selectivity for a count query and the size of the sample for an avg query. For the latter, we also compare variational subsampling to three other methods: central limit theorem (CLT), bootstrap, and traditional subsampling.

For this analysis, we used synthetic queries and datasets to easily control their statistical properties. The attribute values had a mean of 10.0 and a standard deviation of 10.0. To assess the quality of the error estimates, we generated 1,000 independent random samples (each sample was a subset) of the original dataset, recorded the estimated errors based on each random sample, and finally measured three statistics of the estimated errors: mean, 5th, and 95th percentiles.

First, Figure 8a depicts the estimated errors together with the groundtruth relative errors. The sample size, n, was 10K. The groundtruth errors were computed based on our statistical knowledge of

[12] We excluded three TPC-H queries (i.e., tq-3, tq-10, and tq-20) due to SnappyData's failure in creating the samples stratified on extremely high-cardinality columns.



(a) Estimated error for different selectivity

(b) Estimated error for different sample sizes

Figure 8: The accuracy of variational subsampling's error estimation (the error bars are the 5th and 95th percentiles).

the original data. The *relative* errors decreased as the selectivity increased, since the answers to `count` queries themselves were larger with larger selectivities. Overall, variational subsampling's error estimates were within 7% of the groundtruth. The next experiment shows that this deviation is to be expected, due to the properties of random sampling.

Second, Figure 8b compares the quality of variational subsampling's error estimation to that of other methods. Here, the groundtruth values are shown as a reference. When the sample size was small (i.e., 100K), resampling-based techniques were inferior to CLT since we limited the number of resamples (b) to 100. However, the gap reduced with larger samples. Compared to traditional subsampling, variational subsampling was 6.5% less accurate for $n = 100K$ (with selectivity 0.1%); however, the difference decreased to 4.8% for $n = 1M$ and 0% for $n = 10M$. Processing 10M tuples—including the variational subsampling—took only 5.54 seconds.

7 RELATED WORK

Approximate Query Processing — Sampled-based AQP received substantial attention in the research community [8, 9, 11, 13, 14, 17, 20, 20, 22, 25, 26, 30, 32, 33, 41, 50, 52, 53, 55–57, 59, 62, 65, 70]. STRAT [17], AQUA [9], and BlinkDB [11] have used different strategies for creating optimal stratified samples. Online Aggregation (OLA) [19, 31, 54, 69] continuously refines its answers during query execution. In the future, VerdictDB can also adopt some of the techniques proposed in the literature. In this work, however, we focused on variational subsampling, which enabled efficient error estimation for a wide-class of SQL queries without resorting to any tightly-integrated implementations.

Middleware-based Query Rewriting — In our prior work, we have used query rewriting to enforce security policies transparently from the users [21], or to speed up future queries by exploiting past query answers [42, 59]. While Aqua [8] and IDEA [24] have also used query rewriting for AQP, VerdictDB supports a much wider range of queries (including non-PK-FK joins and nested queries), can work with modern distributed query engines (e.g., Hive, Spark, Impala, Redshift), and does not rely on non-SQL code for sample creation. For example, since Aqua relies on CLT-based closed-forms, it requires independent random variables, which means it can only support PK-FK joins. Also, due to Aqua's use of closed-forms, it cannot support UDAs. VerdictDB has overcome this limitation with variational subsampling, which achieves generality without losing efficiency. Furthermore, Aqua relies on the underlying engine's ability to enforce PK-FK relationships, a feature that is missing in most modern SQL-on-Hadoop engines.

Stratified Sample Construction Techniques — BlinkDB [11] constructs stratified samples in two passes: one to count the size of each stratum, and another to perform reservoir sampling for each stratum. Unfortunately, implementing a per-group reservoir sampling in SQL is highly complex. For each stratum, the tuples must be separated, randomly shuffled (i.e., ordered by random integers generated on-the-fly), then filtered using a `limit` clause. The computational cost increases linearly with the number of strata. Quickr [33] constructs stratified samples in one pass. While scanning the table, it counts the number of tuples (for each stratum) that have been read. Based on this count, Quickr's sampler gradually reduces the sampling probability. Implementing this approach in SQL is not straightforward.

8 CONCLUSION

In this paper, we have shown that Universal AQP (i.e., database-agnostic AQP) is a viable approach. We have proposed techniques for sample creation and error estimation that rely solely on standard SQL queries; without making any modifications to existing databases, our AQP solution can operate atop any existing SQL-based engine. Not only is our driver-level solution comparable to fully integrated AQP engines in terms of performance, in some cases it even outperforms them, thanks to its novel error estimation technique, called *Variational Subsampling*. To the best of our knowledge, we are the first to use subsampling in an AQP context. We also proved that, while significantly faster than traditional subsampling, our Variational Subsampling retains the same asymptotic properties, and can handle joins and complex queries. Overall, we demonstrated that VerdictDB offers massive speedups (18.45× on average, and up to 171× with less than 2.6% relative errors) to a variety of popular query engines, including Impala, Spark SQL, and Amazon Redshift.

Future work — We plan to add drivers to support additional databases (Presto, Teradata, Oracle, HP Vertica). Our future research plans include (1) exploring online sampling in a middleware setting, (2) creating a robust physical designer [46] to decide which samples to build, and (3) performing a comprehensive study of how VerdictDB's approximation features affect user behavior.

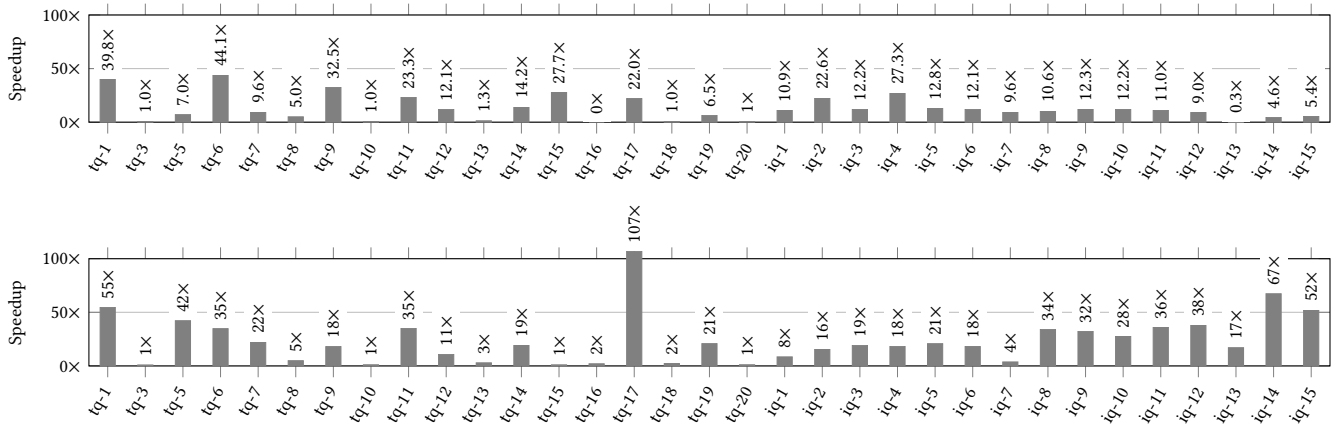


Figure 9: VerdictDB's speedups for Spark (top) and Redshift (bottom).

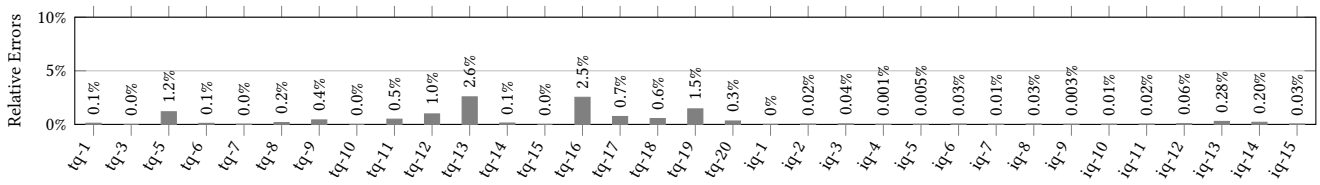


Figure 10: Actual relative errors of the approximate answers (the associated speedups are reported in Figure 4).

A ACKNOWLEDGEMENT

This research is in part supported by National Science Foundation through grants 1629397, 1544844, and 1553169. The authors are grateful to Morgan Lovay, Jeffrey Naughton, and an anonymous reviewer from Google for their insightful comments.

B ADDITIONAL EXPERIMENTS

B.1 Actual Errors of VerdictDB's Answers

This section reports the actual relative errors of VerdictDB's AQP performed in Section 6.2. VerdictDB's ability to estimate those actual errors are separately studied in Section 6.5.

Figure 10 shows the actual relative errors for all 33 queries. The errors were nearly identical across different engines (module negligible differences due to the nature of random sampling); thus, we only report the results for Impala here. The errors were between 0.03%–2.57%. The primary reason for observing different errors was due to the cardinality of the grouping attributes. For example, if there are 10× more unique values in the grouping attributes, the number of tuples averaged by AQP is reduced by 10×, which in turn increases the approximation error by about $\sqrt{10}\times$ (≈ 3.2).

B.2 Sample Preparation Time

In this section, we demonstrate that VerdictDB's sampling preparation is sufficiently fast compared to typical tasks needed for preparing data in cluster. Since the runtime overhead of the ETL process could vary depending on the types of workloads (e.g., from simple csv parsing to entity recognition with natural language processing techniques), we compared VerdictDB's sampling time to the default runtime overhead that must occur: data transfer time. Also, we compare VerdictDB sampling time to SnappyData's sampling time.

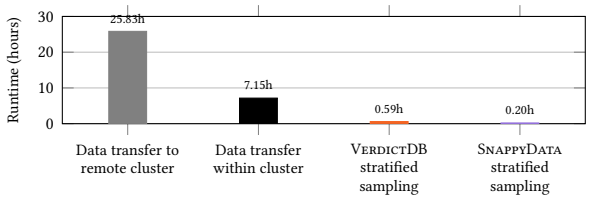
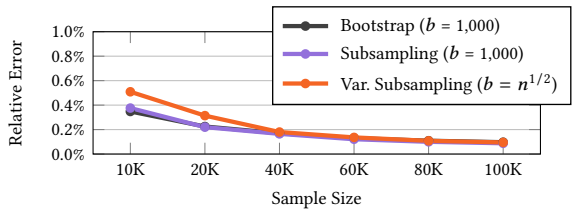


Figure 11: Comparing VerdictDB's sampling time to other data preparation times for the 370 GB dataset.

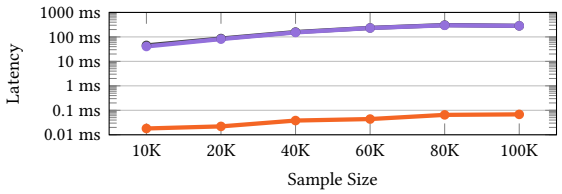
We measured two types of data transfer overhead. The first was the data transfer to a remote cluster (i.e., scp files to an AWS instance). The second was the data transfer within a cluster (i.e., file uploads to HDFS). Figure 11 depicts the results. VerdictDB's sample preparation time was much smaller compared to the other tasks. This is because sampling creation workloads are mostly read-only, which distributed storage systems (e.g., HDFS) support well. The other tasks asked heavy write loads. Even though our cluster had SSD, the runtime was still much slower. SnappyData's sampling was faster than VerdictDB due to its tight integration.

B.3 Further Study of Variational Subsampling

The accuracy and the convergence rate of resampling-based techniques are typically studied under the assumption that the number of resamples, b, is very large (almost infinite). In practice, however,



(a) Accuracy of error bound estimation



(b) Latency of error bound estimation

Figure 12: Time-error tradeoff for different sample sizes (n).

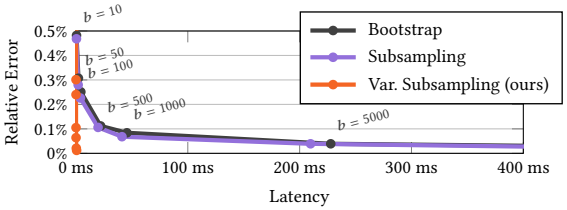


Figure 13: The impact of the number of resamples (b).

the value of b can considerably affect the query performance. In this section, we study this both empirically and theoretically.

Comparison Against Other Techniques — We empirically compared variational subsampling against bootstrap and traditional subsampling, in terms of both accuracy and latency. In general, the accuracy of resampling-based error estimation techniques increases as n and b increase. To verify this, we first varied n (from 10,000 to 100,000), and measured the latency and accuracy of computing an error bound with 95% confidence. We measured accuracy using the relative error with respect to the true mean. For instance, if the true mean was \$100.0, the estimated upper bound was \$110.1, and the true upper bound was \$110.0, then the relative error of the estimated error bound was computed as (|\$110.1 - \$110.0| / \$100.0 * 100)% = 0.1%. The number of resamples (b) was fixed to 1,000 for bootstrap and traditional subsampling. For variational subsampling, b was set to $n^{1/2}$. The results of this experiments are reported in Figures 12a, 12b and 13.

Figure 12a shows that bootstrap produced more accurate error estimates than both traditional and variational subsampling (i.e., the relative errors of the estimated error bounds were lower), but the accuracy gap reduced as n increased. However, as shown in Figure 12b, variational subsampling was orders of magnitude faster than both bootstrap and traditional subsampling for the same sample size. In Figure 13, we also show the relationship between the

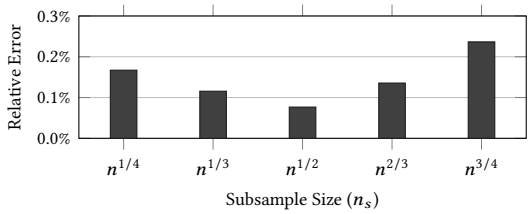


Figure 14: The effect of the subsample size, n_s, on variational subsampling. The sample size was fixed, n=500K.

number of resamples (b) and the relative error of the estimated error bound. Due to the prohibitive costs of bootstrap and traditional subsampling, variational subsampling's relative errors were significantly lower given the same time budget.

Impact of Subsample Size — When the number of resamples b is small, the empirical distribution of b resample-based estimates approximates a sampling distribution. In this case, an additional error term, $O(b^{-1/2})$, must be considered based on the Dvoretzky–Kiefer–Wolfowitz inequality (page 59, [64]).

With a finite b, the error of both traditional subsampling and variational subsampling is in the order of $n_s^{-1/2} + n_s/n + b^{-1/2}$. Since variational subsampling uses $b = n/n_s$ by default, the error term becomes $n_s^{-1/2} + n_s/n + (n/n_s)^{-1/2}$, which can be used to derive an optimal value for n_s. Observe that the second term can simply be ignored since it shrinks faster than the third term. Setting the derivative of $n_s^{-1/2} + (n/n_s)^{-1/2}$ to zero produces $n_s = n^{-1/2}$. In other words, the error expression is minimized when $n_s = n^{-1/2}$.

To empirically validate this choice, we measured the relative errors of the error bound estimates for several choices of n_s, namely $n^{1/4}$, $n^{1/3}$, $n^{1/2}$, $n^{2/3}$ and $n^{3/4}$. Figure 14 shows the results. Here, the sample size, n, was fixed to 50,000. The results show that VerdictDB's default policy (i.e., $n_s = n^{1/2}$) yields the lowest errors.

C PROOFS

In this section, we present the deferred proofs to Lemma 1, Theorem 2, and Theorem 4. For each theorem, we repeat the theorem for convenience and present its proof.

Lemma 1. *Let a sample be constructed by Bernoulli sampling from n tuples with p sampling probability. Then, the sampling probability for outputting at least m tuples with probability $1-\delta$ is*

$$f_m(n) = g^{-1}(m; n)$$

$$\textit{where} \quad g(p; n) = \sqrt{2n \cdot p(1-p)}\ \mathrm{erfc}^{-1}\left(2(1-\delta)\right) + np$$

erfc^{-1} is the inverse of the (standard) complementary error function.

Proof of Lemma 1. Let X denote the number of sampled tuples. Since each tuple is sampled independently with probability p and there are N such tuples, X follows the Binomial distribution $B(N, p)$. We want p to be large enough to satisfy $\Pr(X \geq m) \geq 1-\delta$. With a standard approximation of $B(N, p)$ with a normal distribution $\mathcal{N}(N \cdot p,\ N \cdot p \cdot (1-p))$, we have

$$\int_m^\infty \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x - N \cdot p)^2}{2N \cdot p \cdot (1-p)}\right) \geq 1 - \delta$$

Then,

$$g(p;N) = \sqrt{2N \cdot p(1-p)}\ \mathrm{erfc}^{-1}\left(2(1-\delta)\right) + N\,p \geq m$$
$$p \geq g^{-1}(m;N) \qquad \square$$

Theorem 2. *Let $J_n(x)$ denote the (non-degenerate) true distribution (cumulative distribution function) of the estimate based on a sample of size n. Then, for any n_s such that $n_s \to \infty$ and $n_s/n \to 0$ as $n \to \infty$,*

$$L_n(x) \to J_n(x)$$

in distribution as $n \to \infty$.

Proof of Theorem 2. $L_n(x)$ can be decomposed as follows.

$$L_n(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbf{1}\left(\sqrt{n_{s,i}}(\hat{g}_i - \hat{g}_0) \leq x\right)$$
$$= \frac{1}{b}\sum_{i=1}^{b} \mathbf{1}\left(\sqrt{n_{s,i}}(\hat{g}_i - g) + \sqrt{n_{s,i}}(g - \hat{g}_0) \leq x\right) \qquad (7)$$

Observe that $\frac{1}{b}\sum_{i=1}^{b} \mathbf{1}\left(\sqrt{n_{s,i}}(\hat{g}_i - g) \leq x\right)$ converges to $J_n(x)$ since $n_{s,i} \to \infty$ as $n \to \infty$. Therefore, if the second term, namely $\sqrt{n_{s,i}}(g - \hat{g}_0)$, vanishes to 0, our theorem holds.

According to Hoeffding's inequality,

$$\Pr\left(|\hat{g}_i - E\hat{g}_i| \geq \frac{\varepsilon}{\sqrt{n_{s,i}}}\right) < 2 \cdot \exp(-n \cdot \varepsilon^2/n_{s,i})$$

If $n \cdot \varepsilon^2/n_{s,i} \to \infty$ as $n \to \infty$, then

$$\Pr\left(|\hat{g}_i - E\hat{g}_i| \geq \frac{\varepsilon}{\sqrt{n_{s,i}}}\right) \to 0$$

as $n \to \infty$ for any ε. This means that the second term in Equation 7 converges to 0 in probability as $n \to \infty$.

Thus, we show $n \cdot \varepsilon^2/n_{s,i} \to \infty$ as $n \to \infty$. Observe that $n_{s,i}$ is a binomial random variable $\mathcal{B}(n, 1/\sqrt{n})$. Therefore, the variance of $n_{s,i}/n$ can be expressed as $\left(n \cdot \frac{1}{\sqrt{n}}\left(1 - \frac{1}{\sqrt{n}}\right)\right) \Big/ n = \frac{1}{\sqrt{n}} - \frac{1}{n}$. This variance converges to 0 as $n \to \infty$. Since the variance converges to 0, the probability that $n_{s,i}/n$ is arbitrarily close to 0 is 1.0. Therefore, $n_{s,i}/n \to 0$ in probability.[13] As stated above, this implies that the second term in Equation 7 converges to 0 as $n \to \infty$.

Since $b \to \infty$ as $n \to \infty$, $L_n(x)$ becomes an empirical distribution using an infinite number of samples. Thus, it converges to the true distribution. $\square$

Theorem 4. *If there exists*

$$h(i,j) = k \quad \text{for } (i,j) \in (\mathcal{I} \times \mathcal{J})_k = \mathcal{I}_k \times \mathcal{J}_k \qquad (8)$$

then

$$(T \bowtie S)_v = \Pi_{*,\ h(i,j)\ \mathtt{as\ sid}}\ (T_v \bowtie S_v) \qquad (9)$$

[13] "x converges in probability to y" means that the probability of the absolute difference between x and y being larger than any $\varepsilon > 0$ converges to 0.

Proof of Theorem 4. From Equation 3,

$$(T \bowtie S)_{v,k} = \cup_{i \in \mathcal{I}_k} T_{v,i} \bowtie \cup_{j \in \mathcal{J}_k} S_{v,j}$$
$$= \bigcup_{(i,j) \in (\mathcal{I} \times \mathcal{J})_k} T_{v,i} \bowtie S_{v,j}$$
$$= \sigma_{h(i,j)=k}\,(T_v \bowtie S_v)$$

where $\sigma_{h(i,j)=k}$ is the selection operator.

Based on the above equation, the variational table of the join, namely $(T \bowtie S)_v$, can be expressed as

$$(T \bowtie S)_v = \bigcup_{k=1,\ldots,b} (T \bowtie S)_{v,k}$$
$$= \bigcup_{k=1,\ldots,b} \sigma_{\mathtt{sid}=k}\ \left(\Pi_{*,\ h(i,j)\ \mathtt{as\ sid}}\ (T_v \bowtie S_v)\right)$$
$$= \Pi_{*,\ h(i,j)\ \mathtt{as\ sid}}\ (T_v \bowtie S_v)$$

$\square$

Theorem 3. *Let $g(T,S)$ be an aggregate function involving two tables T and S, and $\hat{g}(T_s, S_s)$ be an estimator of $g(T,S)$, where T_s and S_s are respective samples of T and S. Furthermore, let $T_{s,i}$ and $S_{s,j}$ be the i-th and the j-th subsamples of T_s and S_s, respectively. Lastly, let $n_{s,i,j}$ denote the cardinality of the join of $T_{s,i}$ and $S_{s,j}$. If $|T_s|/|T_{s,i}| = |S_s|/|S_{s,j}|$,*

$$L_n(x) = \frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b} \mathbb{1}\left(\sqrt{n_{s,i,j}}\left(\hat{g}(T_{s,i}, S_{s,j}) - \hat{g}(T_s, S_s)\right) \leq x\right)$$

converges to the true distribution $J_n(x)$ of $\hat{g}(T_s, S_s)$ as $n \to \infty$.

Proof of Theorem 3. Define

$$U_n(x) = \frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b} \mathbb{1}\left(\sqrt{n_{s,i,j}}\left(\hat{g}(T_{s,i}, S_{s,j}) - g(T,S)\right) \leq x\right) \qquad (10)$$

To show that $L_n(x)$ converges to the true distribution $J_n(x)$, it suffices to show that the above U-statistic, i.e., $U_n(x)$, converges to the true distribution of $\hat{g}(T_s, S_s)$ (see the proof of theorem 2.1 in [61]). When Equation 10 involves subsamples of a single sample, Hoeffding's inequality for U-statistics can be applied to show the convergence of $U_n(x)$ to the true distribution. However, since Equation 10 involves respective subsamples of T_s and S_s, Hoeffding's inequality is not directly applicable.

To show the convergence of $U_n(x)$ to the distribution $J_n(x)$ of $\hat{g}(T_s, S_s)$, we employ the result on two-sample statistics [37]. This result indicates that, if the value of $U_n(x)$ does not depend on the orders of the sampled tuples, $\sqrt{n}(U_n(x) - E(U_n(x)))$ is asymptotically normally distributed (with mean zero), as $|T_s| \to \infty$. Note that $U_n(x)$ is an unbiased estimator of the true distribution; thus, $E(U_n(x))$ is simply $J_n(x)$.

An implication of the above result is that the variance of $\sqrt{n}(U_n(x) - E(J_n(x)))$ is finite. This implies that, for any x, $\mathrm{Var}(U_n(x) - J_n(x)) \to 0$ as $n \to \infty$. Thus, $U_n(x)$ converges to $J_n(x)$ as $n \to \infty$.

$\square$

REFERENCES

[1] Fast, approximate analysis of big data (yahoo's druid). http://yahooeng.tumblr.com/post/135390948446/data-sketches.

[2] Infobright approximate query (iaq). https://infobright.com/introducing-iaq/.

[3] Instacart Orders, Open Sourced. https://www.instacart.com/datasets/grocery-shopping-2017. Accessed: 2017-09-17.

[4] Presto: Distributed SQL query engine for big data. https://prestodb.io/docs/current/release/release-0.61.html.

[5] SnappyData Inc. http://snappydata.io.

[6] TPC-H Benchmark. http://www.tpc.org/tpch/.

[7] *Comparing Recent Approaches for Bootstrapping Sample Survey Data: A First Step Toward a Unified Approach*, 03 2013. Joint Statistical Meeting (JSM) 2012, San Diego, CA, 2012.

[8] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

[9] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *SIGMOD*, 1999.

[10] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[11] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[12] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. *PVLDB*, 2012.

[13] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, 2004.

[14] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.

[15] P. J. Bickel, F. Götze, and W. R. van Zwet. Resampling fewer than n observations: gains, losses, and remedies for losses. In *Selected Works of Willem van Zwet*. 2012.

[16] A. J. Canty, A. C. Davison, D. V. Hinkley, and V. Ventura. Bootstrap diagnostics and remedies. *Canadian Journal of Statistics*, 34(1), 2006.

[17] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.

[18] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, 1999.

[19] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[20] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[21] K. Eykholt, A. Prakash, and B. Mozafari. Ensuring authorized updates in multi-user database-backed applications. In *USENIX Security Symposium*, 2017.

[22] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu. Querying big data by accessing small data. In *PODS*, 2015.

[23] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[24] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 2017.

[25] V. Ganti, M.-L. Lee, and R. Ramakrishnan. Icicles: Self-tuning samples for approximate query answering. In *VLDB*, 2000.

[26] W. Gatterbauer and D. Suciu. Approximate lifted inference with probabilistic databases. *PVLDB*, 2015.

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

[28] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 2008.

[29] P. Hall. On symmetric bootstrap confidence intervals. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50, 1988.

[30] W. He, Y. Park, I. Hanafi, J. Yatvitskiy, and B. Mozafari. Demonstration of VerdictDB, the platform-independent AQP system. In *SIGMOD*, 2018.

[31] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[32] K. Hose, D. Klan, and K.-U. Sattler. Distributed data summaries for approximate query processing in pdms. In *IDEAS*, 2006.

[33] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, 2016.

[34] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I. Jordan. A general bootstrap performance diagnostic. In *KDD*, pages 419–427, 2013.

[35] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan. The big data bootstrap. In *ICML*, 2012.

[36] N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 5(10):1028–1039, 2012.

[37] E. L. Lehmann. Consistency and unbiasedness of certain nonparametric tests. *The Annals of Mathematical Statistics*, 1951.

[38] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 2015.

[39] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016.

[40] D. G. Mayo. In defense of the neyman-pearson theory of confidence intervals. *Philosophy of Science*, 1981.

[41] A. Meliou, C. Guestrin, and J. M. Hellerstein. Approximating sensor network queries using in-network summaries. In *IPSN*, 2009.

[42] B. Mozafari. Verdict: A system for stochastic query planning. In *CIDR, Biennial Conference on Innovative Data Systems*, 2015.

[43] B. Mozafari. Approximate query engines: Commercial challenges and research opportunities. In *SIGMOD*, 2017.

[44] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.

[45] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[46] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[47] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.

[48] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. SnappyData: A unified cluster for streaming, transactions, and interactive analytics. In *CIDR*, 2017.

[49] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *PVLDB*, 8, 2014.

[50] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, 2010.

[51] S. Nirkhiwale, A. Dobra, and C. Jermaine. A sampling algebra for aggregate estimation. *PVLDB*, 2013.

[52] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive Analysis of Web-Scale Data. In *CIDR*, 2009.

[53] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, 2010.

[54] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.

[55] Y. Park. Active database learning. In *CIDR*, 2017.

[56] Y. Park, M. Cafarella, and B. Mozafari. Neighbor-sensitive hashing. *PVLDB*, 2015.

[57] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. *ICDE*, 2016.

[58] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. VedictDB: Universalizing approximate query processing. https://arxiv.org/2215346, 2018.

[59] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database Learning: Towards a database that becomes smarter every time. In *SIGMOD*, 2017.

[60] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *SIGMOD*, 2005.

[61] D. N. Politis and J. P. Romano. Large sample confidence regions based on subsamples under minimal assumptions. *The Annals of Statistics*, 1994.

[62] N. Potti and J. M. Patel. Daq: a new paradigm for approximate query processing. *PVLDB*, 2015.

[63] J. Ramnarayan, B. Mozafari, S. Menon, S. Wale, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. SnappyData: A hybrid transactional analytical store built on spark. In *SIGMOD*, 2016.

[64] R. J. Serfling. *Approximation theorems of mathematical statistics*. 2009.

[65] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR*, 2011.

[66] R. L. Smith. Extreme value theory. *Handbook of applicable mathematics*, 1990.

[67] H. Su, M. Zait, V. Barrière, J. Torres, and A. Menck. Approximate aggregates in oracle 12c, 2016.

[68] S. Vrbsky, K. Smith, and J. Liu. An object-oriented semantic data model to support approximate query processing. In *Proceedings of IFIP TC2 Working Conference on Object-Oriented Database Semantics*, 1990.

[69] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD*, pages 651–662, 2010.

[70] F. Xu, C. Jermaine, and A. Dobra. Confidence bounds for sampling-based group by estimates. *TODS*, 2008.

[71] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, 2015.

[72] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. ABS: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[73] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.