Database Learning: Toward a Database that Becomes Smarter Every Time

Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, Barzan Mozafari
University of Michigan, Ann Arbor
{pyongjoo,tajik,michjc,mozafari}@umich.edu

ABSTRACT

In today's databases, previous query answers *rarely* benefit answering future queries. For the first time, to the best of our knowledge, we show how we can completely change this paradigm in an approximate query processing (AQP) context. We make the following observation: the answer to each query reveals some degree of *knowledge* about the answer to another query because their answers stem from the same underlying distribution that has produced the entire dataset. Exploiting and refining this underlying knowledge should allow us to answer queries more analytically, rather than by reading enormous amounts of raw data. Also, processing more queries should continuously enhance our knowledge of the underlying distribution, leading to faster processing of future queries.

We call this novel idea—learning from past query answers—*Database Learning*. We exploit *the principle of maximum entropy* to produce answers guaranteed to always be more accurate than existing sampling-based approximations. Empowered by this idea, we build a query engine atop Spark SQL, called Intelli. We conduct extensive experiments on real-world query traces from a large customer of a major database vendor. Our results demonstrate that database learning supports 73.7% of these queries, speeding them up by up to 10.43x compared to existing AQP systems. Database learning therefore provides a revolutionary and generic means of reusing work in a database: it allows a DBMS to *continuously learn and improve*; the more queries issued in the past, the faster and more accurate answers in the future.

1. INTRODUCTION

In today's databases, the answer to a previous query is rarely useful for speeding up new queries. Besides a few limited benefits (see Previous Approaches below), the work (both I/O and computation) performed for answering past queries is often wasted afterwards. However, in an approximate query processing context (e.g., SnappyData [4], Presto [3], Druid [2], BlinkDB [9]), one might be able to change this paradigm altogether and re-use much of the previous work done by the database based on the following observation:

> *The answer to each query reveals some* fuzzy knowledge *about the answers to other queries, even if each query accesses a different subset of tuples and columns.*

This is because the answers to different queries stem from the same underlying distribution which has produced the entire dataset. In other words, **each answer reveals a piece of information about this underlying but unknown distribution**. Note that having a concise statistical model of the underlying data can have significant performance benefits. In the ideal case, if we had access to an incredibly precise model of the underlying data, we would no longer have to access the data itself. In other words, we could answer queries more efficiently by analytically evaluating them on our concise model, which would mean reading and manipulating a few kilobytes of model parameters rather than terabytes of raw data. While we may never have a perfect model in practice, even an imperfect model can be quite useful. Instead of using the entire data, one can use a small sample of it to quickly produce a sample-based approximate answer, which can be then calibrated and combined with the model to produce a more accurate approximate answer to the query. **The more precise our model, the less need for actual data, the smaller our sample, and consequently, the faster our response time.** In particular, if we could somehow continuously improve our model—say, by *learning* a bit of information from every query and its answer—we should be able to **answer new queries using increasingly smaller portions of data, i.e., become smarter and faster as we process more queries.**

We call the above goal *Database Learning* (DBL), as it is reminiscent of the inferential goal of machine leaning (ML), where past observations are used to improve future predictions [15, 43]. Likewise, our goal in DBL is to enable a similar principle by **learning from past observations, but in a query processing setting**. Specifically, in DBL, we plan to treat approximate answers to past queries as observations, and use them to update our posterior knowledge of the underlying data, which in turn can be used to speed up future queries.

In Figure 1, we visualize this idea using a toy example. Here, DBL learns a model to explain the average sales for different times of the year. Figure 1(a) shows this model based on the answer to the first query. Since the model is probabilistic, its 95% confidence interval is also shown (the

WOODSTOCK '97 El Paso, Texas USA

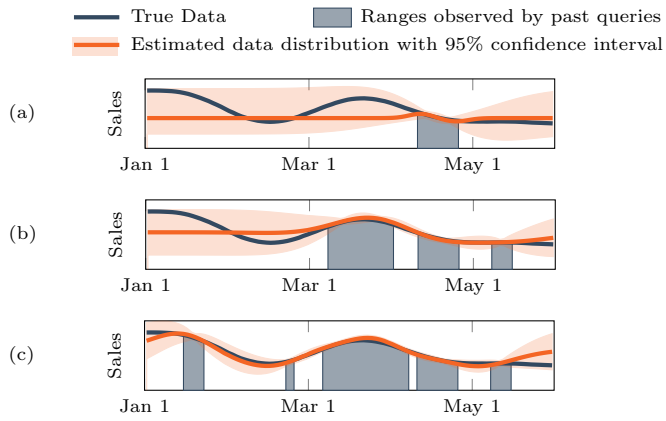
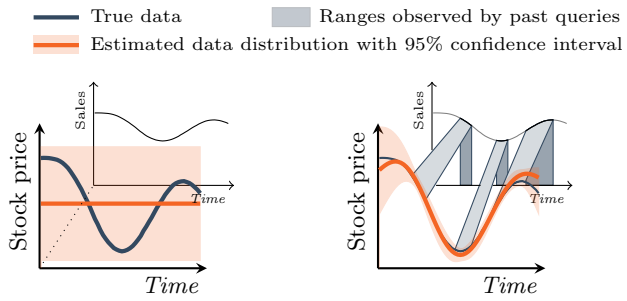


Figure 1: An example of how database learning might continuously refine its model as more queries are processed: (a) after 1 query, (b) after 2 queries, and (c) after 5 queries. A similar learning process can be used for categorical data. Also, database learning can be effective even when its model does not exactly coincide with the ground-truth data.



(a) Our estimation of the company's average stock price over time, prior to running any queries.

(b) Our estimation of the company's average stock price might improve even if we query sales over time.

Figure 2: In DBL, queries may still benefit one another even if they access different columns of the data.

shaded area around the best current estimate). As shown in Figure 1(b-c), DBL further refines its model every time a new query is answered. This approach allows a DBL-enabled query engine to provide increasingly more accurate estimates of average sales *even for time ranges that have never been accessed by previous queries*—this is possible because DBL finds the most likely model of the entire area that fits with the past query answers. This example is to illustrate the possibility of (i) significantly faster response times by processing smaller samples of the data for the same answer quality, or (ii) increasingly more accurate answers for the sample size and response time.

Interestingly, a similar approach can be taken even if the queries access different columns. Figure 2 shows an example of how DBL's estimation of a company's average stock price might improve as a result of querying the company's sales. This is because any non-zero correlation between a company's average sales and stock price would mean that, the likelihood of observing certain values for one should change based on the values observed for the other.

Therefore, DBL can be a revolutionary leap in database technology and approximate query processing: from today's DBMSs where previous work is rarely re-used to a new world where a DBMS *continuously learns and improves*; the more queries issued in the past, the faster and more accurate answers in the future.

Challenges — To realize DBL's vision, three key challenges must be overcome in practice. First, there is a *query generality* challenge. DBL must be able to transform a wide class of SQL queries into appropriate mathematical representations so that they can be used by statistical methods for improving new queries. Second, there is a *data generality* challenge. To support arbitrary datasets, no distributional assumptions must be made about the underlying data. In other words, the only valid knowledge must come from past queries and their respective answers. Finally, there is an *efficiency* challenge. We need to strike a balance between the computational complexity of our inference and its ability to reduce the error of query answers. In other words, DBL needs to be both effective and practical.

Previous Approaches — In today's databases, the work performed for answering past queries is rarely beneficial to new queries, except for the following cases:

1. **View selection and adaptive indexing**: In predictable workloads, columns and expressions commonly used by past queries provide hints on which indices [22, 26, 38] or materialized views [11] to build.
2. **Caching**: The recently accessed tuples might still be in memory when future queries access the same tuples.

Both techniques, while beneficial, can only reuse previous work to a limited extent. Caching input tuples reduces I/O if data size exceeds memory, but does not reuse query-specific computations. Caching (intermediate) final results can reuse computation only if future (sub-)queries are *identical* to those in the past. While index selection techniques use the knowledge of which columns are commonly filtered on, an index per se does not allow for reusing computation from one query to the next. Adaptive indexing schemes (e.g., database cracking [26]) use each query to incrementally refine an index, to amortize the cost across queries. However, there is still an exponential number of possible column-sets that can be indexed. Also, they do not reuse query-specific computations either. Finally, materialized views[1] are only beneficial when there is a strict structural compatibility—such as query containment or equality—between past and new queries [23].

While orthogonal to these existing techniques, DBL is fundamentally different:

1. Unlike indices and materialized views, DBL incurs *little storage overhead* as it only retains the past n aggregate queries and their answers.[2] Consequently, indices and materialized views grow in size as the data grows, while DBL remains *oblivious to the data size*.
2. Materialized views, indexing, and caching are exact methods and thus are only effective when new queries touch previously accessed columns or tuples. DBL is strictly *more general* as it can benefit new queries even if they require tuples that were not touched by past queries. This is due to DBL's probabilistic model,

[1] **DBL can be easily misunderstood with view selection**. Not only do they take fundamentally different approaches (statistical versus exact), they also differ in generality and other aspects, as explained next.

[2] Even if a query outputs too many tuples, DBL retains only a fixed number of them (see Section 2.4)

which provides extrapolation power spanning the entire data (see Figure 1).

Our Approach — Note that our vision of database learning (DBL) might be achieved in different ways, depending on the design decisions made in terms of query generality, data generality, and efficiency. In this paper, besides the introduction of the concept of DBL, we also provide a specific solution for achieving DBL, which we call INTELLI to distinguish it from DBL as a general vision.

From a high-level, INTELLI overcomes the challenges associated with query generality, data generality, and efficiency, as follows. First, complex SQL queries are decomposed into simpler *snippets*. The answer to each snippet is then modeled as a probabilistic random variable, corresponding to an integration over relevant tuples drawn from an unknown underlying distribution. Second, to achieve data generality, we employ a *non-parametric* probabilistic model, whereby answers to different snippets are related via a joint probability distribution function (pdf). We derive this joint pdf by exploiting a powerful statistical principle, namely the *principle of maximum entropy* [46], which yields the *most likely* pdf given our limited statistical knowledge (i.e., past queries and their approximate answers). Third, to ensure the computational efficiency of our system, we restrict ourselves to only the first and second-order statistics of the query answers (i.e., mean, variance, and covariance) when applying the principle of maximum entropy. We show that this instantiation of DBL leads to significant speedup of query processing.

Contributions — In this paper, we make the following contributions:

1. We introduce the novel concept of *database learning* (DBL), which goes beyond all previous techniques in reusing previous work in a DBMS. By learning from past query answers, DBL allows DBMS to continuously become smarter and faster at answering new queries.
2. We provide a concrete instantiation of DBL, called INTELLI, using the principle of maximum entropy and a *kernel*-based statistical modeling technique. INTELLI's strategies cover 63.6% of TPC-H queries and 73.7% of a real-world query trace from a major customer of HP Vertica (a leading vendor of analytical DBMS). We formally show that INTELLI's answers are never worse than existing AQP techniques.
3. We integrate INTELLI atop an open-source query engine and conduct extensive experiments using both benchmark and real-world traces, showing upto 10.43x speedup and 98% error reduction compared to existing AQP engines that do not use INTELLI.

The rest of this paper is organized as follows. Section 2 overviews INTELLI's workflow, supported query types, and query processing. Sections 3-6 describe the internals of INTELLI in detail. Section 7 reports our empirical results. Section 8 summarizes related work, and Section 9 concludes the paper with future work.

2. SYSTEM OVERVIEW

In this section, we overview the system we have built based on database learning (DBL), called INTELLI. Section 2.1 explains INTELLI's architecture and overall workflow. Section 2.2 describes supported SQL query types. Sections 2.3 and 2.4 overview INTELLI's computation and query representation, respectively. Lastly, Section 2.5 discusses INTELLI's current limitations.

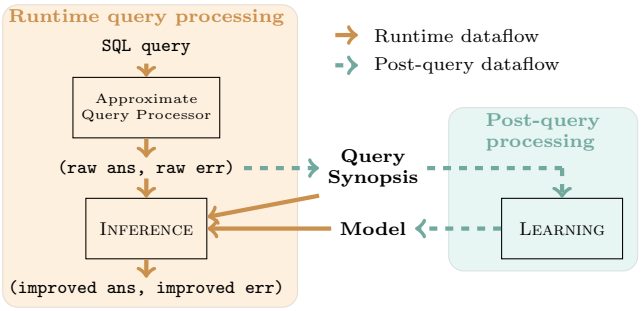


Figure 3: Workflow in INTELLI. At query time, the INFERENCE module improves a query answer obtained by an underlying AQP (i.e., *raw answer*) using a *Query Synopsis* and a *Model*. Each time a query is processed, the raw query answer, `(raw ans, raw err)`, is added to the query synopsis. The LEARNING module uses this updated query synopsis to refine the current model accordingly.

2.1 Architecture

INTELLI consists of a *query synopsis*, a *model*, and three processing modules (an off-the-shelf approximate query processing engine, an INFERENCE module, and a LEARNING module). Figure 3 illustrates the connection between these different componenets.

The *query synopsis* contains a summary of past queries[3] and their approximate answers computed by an underlying off-the-shelf AQP (approximate query processing) engine. The query synopsis is initially empty when INTELLI is first launched. Once the i-th query is processed, if it is a supported query (defined in Section 2.2), INTELLI adds a triplet (q_i, θ_i, β_i) to the query synopsis, where q_i is the i-th query, θ_i is an (approximate) answer to q_i, and β_i is the estimated error for θ_i. The θ_i and β_i are obtained from the underlying AQP engine.[4] We call the queries stored in the query synopsis *past queries*.

The second key component is a *model* representing INTELLI's statistical understanding of the underlying data. The model is trained on the *query synopsis*, and is updated each time a query is added to the synopsis.

At query time, for an incoming query (which we call a *new query*), INTELLI invokes the AQP engine to compute a pair of an approximate answer θ_i and an estimated error β_i for the new query, called the *raw answer* and the *raw error*, respectively. Then, INTELLI combines this raw answer and the previously trained model to *infer* an *improved answer* and an updated error estimate, called the *improved error*. We prove that the improved error is never larger than the raw error (Theorem 1). Non-aggregate expressions are left unmodified by INTELLI. Table 1 summarizes the terminology used in this paper.

2.2 Supported Queries

INTELLI supports aggregate queries that are flat (i.e., no derived tables or sub-queries) with the following conditions:

[3]DBL is focused on analytical (i.e., aggregate) queries only. Thus, we use 'analytical queries' and 'queries' interchangeably in this paper.
[4]For simplicity, and without loss of generality, we assume that θ_i and β_i are computed using *a sampling-based approximate query engine*, e.g., BlinkDB [10], ODM [40], ABS [49], DBO [19], or SnappyData [4].

Term	Definition
INTELLI	our actual system based on database learning.
true answer	the exact answer.
raw answer	the answer computed by the AQP engine.
raw error	the estimated error by the AQP engine.
improved answer	the answer updated by INTELLI.
improved error	the estimated error by INTELLI.
past query	a supported query processed in the past.
new query	an incoming query whose answer is to be computed.

Table 1: Terminology.

1. **Aggregates**. Any number of `SUM`, `COUNT`, or `AVG` aggregates can appear in the `select` clause. The arguments to these aggregates can also be a *derived attribute*.
2. **Joins**. Foreign-key joins between a fact table and any number of dimension tables are supported. For simplicity, our discussion in this paper is based on a denormalized table.
3. **Selections**. INTELLI currently supports equality and inequality comparisons for categorical and numerical attributes (including the `in` operator). Currently, INTELLI does not support any disjunctions or textual filters (e.g., `like '%Apple%'`) in the `where` clause.
4. **Grouping**. `groupby` clauses are supported for both stored and derived attributes. The query may also include a `having` clause.[5]

Nested Query Support — Although INTELLI does not directly support nested queries; many queries can be flattened using joins [1] or by creating intermediate views for sub-queries [23]. In fact, this is the process used by Hive for supporting the nested queries in TPC-H benchmark [29]. We are currently working to automatically process nested queries and to expand the class of supported queries (see Section 9).

Unsupported Queries — Upon its arrival, each query is inspected by INTELLI's query type checker to determine whether it can be supported, and if not, INTELLI bypasses the INFERENCE module and simply returns the raw answer. The overhead of the query type checker is negligible compared to the runtime of the AQP engine; thus, INTELLI does not incur any noticeable runtime overhead even when a query is not supported.

Only supported queries are stored in INTELLI's query synopsis and used to improve the accuracy of answers to future supported queries; that is, the class of queries that can be improved is equivalent to the class of queries that can be used to improve other queries.

2.3 Inference Overview

In this section, we provide the high-level intuition behind our approach. Consider a relation whose tuples are drawn from some unknown underlying distribution. We represent the possible values of these tuples as well as different aggregations of any subset of them using random variables. INTELLI's INFERENCE module is entirely based on this random variable interpretation of tuples and query answers.

For instance, let $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ represent the possible query answers to two queries, q_1 and q_2, and suppose we want to estimate the true answer $\bar{\theta}_3$ to q_3 using the actual values of $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, denoted by θ_1 and θ_2.[6] Let $\bar{\boldsymbol{\theta}}_3$ be the random variable representing the possible values for $\bar{\theta}_3$. Then, our goal becomes finding the most likely value for $\bar{\boldsymbol{\theta}}_3$.

To solve this problem, INTELLI first expresses the relationship among the three random variables ($\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, and $\bar{\boldsymbol{\theta}}_3$) using a joint probability distribution function (pdf), say $f(\theta_1, \theta_2, \bar{\theta}_3)$. Intuitively, this joint pdf encodes the chance that an AQP engine outputs that particular combination of values as query answers to these three queries. Determining this joint pdf is a challenging task given that we cannot directly inspect the unknown underlying distribution that has generated the tuples in the relation. INTELLI therefore overcomes this challenge by applying the principle of maximum entropy given certain statistics derived from past queries and their answers.

Conceptually, once this joint pdf is determined, the most likely outcome of $\bar{\boldsymbol{\theta}}_3$ can be estimated by computing a conditional pdf, i.e., $f(\bar{\theta}_3 \mid \boldsymbol{\theta}_1 = \theta_1, \boldsymbol{\theta}_2 = \theta_2)$, and then finding a $\bar{\theta}_3$ for which the conditional pdf takes its maximum value. INTELLI returns this value as an improved answer. The inference processed will be presented in more detail in Section 3.

2.4 Internal Representation

Query Synopsis — When a query (along with its raw answer and raw error) are inserted into the query synopsis, it is broken into multiple individual records. We call each of those records a *snippet*. Conceptually, each snippet corresponds to a supported SQL query with a single aggregate function and no other projected columns in its `select` clause, and with no `groupby` clause; thus, the answer to each snippet is a single real number. A SQL query with multiple aggregate functions or a `groupby` clause is converted to a set of multiple snippets as follows:

1. If the query's `select` clause contains N^{agg} aggregate functions, INTELLI creates N^{agg} snippets, identical to the original query but each with only one of its aggregate functions. Non-aggregate columns are removed from the projection list.
2. If the query has a `groupby` clause, N^{grp} snippets are created, where N^{grp} is the number of groups generated by the query.[7] However, instead of the `groupby` clause, a new predicate is added to the `where` clause of each snippet, with equality filters corresponding to one of the N_r combinations of column values.

A potential challenge of the above approach is that the number of generated snippets, $N^{agg} \times N^{grp}$, can be extremely large, e.g., if a `groupby` clause includes a primary key. To ensure that the number of snippets added per each snippet is bounded, INTELLI only generates snippets for the first N^{max} groups in the answer set. We find that $N^{max} = 1000$ works quite well in practice [36].

Moreover, for each aggregate function g, the query synopsis a maximum of C_g snippets by following a least recently

[5]Note that the underlying AQP engine may affect the cardinality of the result set depending on the `having` clause (i.e., subser/superset error [33]). INTELLI simply operates on the result set returned by the AQP engine.

[6]In this paper, we use the bold font $\mathbf{x}$ for a random variable and the regular font x for a specific value taken by that random variable.

[7]Since queries are added after their execution, N^{grp} is known.

used snippet replacement policy (by default, C_g=100 [36]). This improves the efficiency of the inference process, while maintaining an accurate model based on the recently processed query answers.

Since INTELLI's inference and learning processes work with snippets, for ease of presentation we use *query* to mean *snippet* in the rest of this paper.

Aggregate Computation — INTELLI uses two aggregate functions to perform its internal computations: `AVG(`A_k`)` and `FREQ(*)`. The first one is an average function over a given attribute, and the second one is the fraction of tuples in a relation that satisfy a query's selection predicates. As stated earlier, the attribute A_k can be either a stored attribute (e.g., `revenue`) or a derived one (e.g., `revenue * discount`). At runtime, INTELLI combines these two types of aggregates to compute its supported aggregate functions as follows:

- `AVG(`A_k`)` = `AVG(`A_k`)`
- `COUNT(*)` = `FREQ(*)` × (table cardinality)
- `SUM(`A_k`)` = `AVG(`A_k`)` × `COUNT(*)`

2.5 Limitations

INTELLI relies on an off-the-shelf underlying AQP engine for obtaining raw answers and errors. Consequently, INTELLI is bound by the limitations of the AQP engine. For example, sampling-based engines are not apt at supporting arbitrary joins or `MIN`/`MAX` aggregates. Similarly, INTELLI's error guarantees are contingent upon the validity of the AQP engine's error estimates. (However, off-the-shelf error diagnostic techniques can be used [8].)

Second, similar to most asymptotic analyses, the convergence rate INTELLI's inference (i.e., number of required observations) depends on the smoothness of the aggregated values' pdf (probability distribution function). In most cases, one expects that tuples with similar values for many of their attributes be also somewhat correlated in their aggregate values. Informally, smoothness is the extent of this correlation [41], e.g., `MIN`/`MAX` aggregates often lack this property. The smoother the pdf, the fewer queries are needed for inferring the underlying data characteristics. However, even for non-smooth pdfs, we prove that INTELLI never worsens the original raw answers (Theorem 1). We also empirically study INTELLI's effectiveness for different data and query distributions in Section 7.6.

3. INFERENCE

In this section, we describe our inference process for computing an improved answer (and improved error) for a new query. First, we formally state the problem in Section 3.1. To solve this problem, we apply the principle of maximum entropy to capture the relationship between query answers using a joint probability distribution (Section 3.2). Then, we exploit the answers to past queries to conditionalize this probability distribution and infer improved answers to new queries (Section 3.3).

3.1 Problem Statement

Let r be a relation[8] drawn from some unknown underlying distribution. Let r's attributes be $A_1, \ldots, A_m$, where $A_1, \ldots, A_l$ are the *dimension attributes* and $A_{l+1}, \ldots, A_m$ are the *measure attributes*. Dimension attributes cannot appear inside aggregate functions while measure attributes can. Dimension attributes can be numerical or categorical, but measure attributes are numerical. Measure attributes can also be *derived attributes*. Let g be an aggregate function on A_k (e.g., `AVG(`A_k`)`), where A_k is one of the measure attributes, and $\boldsymbol{t}$ be a vector of the values $(a_1, \ldots, a_l)$ for $A_1, \ldots, A_k$.

[8] r can be a join or Cartesian product of multiple tables.

Sym.	Meaning
q_i	i-th (supported) query to database learning
$n+1$	index number for a new query
$\boldsymbol{\theta}_i$	random variable for possible raw answers to q_i
θ_i	(actual) raw answer to q_i; the outcome of $\boldsymbol{\theta}_i$
β_i	raw error (standard deviation) associated with θ_i
$\bar{\boldsymbol{\theta}}_i$	random variable for possible true answers to q_i
$\bar{\theta}_i$	(actual) true answer to q_i; the outcome of $\bar{\boldsymbol{\theta}}_i$

Table 2: Mathematical Notations.

Given a query q_i on r, an approximate query processing (AQP) engine returns an approximate answer θ_i along with an estimated error β_i. Since the AQP engine is aggregating a subset of tuples in r, we can encode the possible answers to q_i as a random variable $\boldsymbol{\theta}_i$, and its actual answer to q_i as θ_i (i.e., a value assigned to $\boldsymbol{\theta}_i$). We use $\bar{\theta}_i$ to denote the true answer to q_i, which would be obtained if we processed the entire dataset. Thus, we can formally define the error β_i using the following equation:

$$\beta_i^2 = E[(\boldsymbol{\theta}_i - \bar{\theta}_i)^2]$$

The AQP engine can trade its answer accuracy for faster response times, i.e., the larger β_i, the faster it is to obtain θ_i.

Suppose INTELLI's query synopsis contains n records, namely Q_n=$\{(q_1, \theta_1, \beta_1), \ldots, (q_n, \theta_n, \beta_n)\}$. Let q_{n+1} be the *new query*. First, the AQP engine computes a raw answer and its raw error for q_{n+1}, as $(\theta_{n+1}, \beta_{n+1})$. Then, DBL computes an improved answer $\hat{\theta}_{n+1}$ and its improved error $\hat{\beta}_{n+1}$ using Q_n and $(\theta_{n+1}, \beta_{n+1})$.

With this notation, our problem is stated as follows. Given Q_n and $(\theta_{n+1},\ \beta_{n+1})$, compute a $(\hat{\theta}_{n+1}, \hat{\beta}_{n+1})$ such that $\hat{\beta}_{n+1} \ll \beta_{n+1}$.

3.2 Relationship among Query Answers

To compute $\hat{\beta}_{n+1}$ using Q_n, we first capture the relationship among possible query answers. Specifically, the relationship is expressed by a joint probability distribution function (pdf) $f(\theta_1, \ldots, \theta_n, \bar{\theta}_n)$ associated with random variables $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n, \bar{\boldsymbol{\theta}}_n)$.

To estimate this joint pdf, INTELLI relies on the principle of maximum entropy (ME) [46], a simple but powerful statistical tool for determining a pdf of random variables given a certain amount of statistical information available. The ME principle states that, subject to some testable information on random variables associated with a pdf in question, the pdf that best represents the current knowledge is the one maximizes the following expression, called *entropy*:

$$h(f) = -\int f(\vec{\theta}) \cdot \log f(\vec{\theta})\, d\vec{\theta} \qquad (1)$$

where $\vec{\theta} = (\theta_1, \ldots, \theta_n, \bar{\theta}_n)$.

Note that, according to the principle of ME, different amounts of statistical information on our random variables

result in different pdfs. In fact, there are two conflicting considerations when applying this principle. On one hand, the resulting pdf can be computed more efficiently if the provided statistics are simple or few, i.e., simple statistics reduce the computational complexity. On the other hand, the resulting pdf can describe the relationship among the random variables more accurately if richer statistics are provided, i.e., the richer the statistics, the better our improved answers. Therefore, we need to choose an appropriate degree of statistical information to strike a balance between the computational efficiency of pdf evaluation and its accuracy in describing the relationship among query answers.

To strike this balance, INTELLI's INFERENCE module relies only on the first and the second order statistics of the random variables, i.e., mean, variances, and covariances. With this choice of statistics, a well-known result in information theory [46] guarantees that the resulting pdf will be a multivariate normal distribution with the corresponding mean, variance, and covariance values.

Lemma 1. Let $\vec{\boldsymbol{\theta}} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}, \bar{\boldsymbol{\theta}}_{n+1})$ be a vector of $n+2$ random variables with mean values $\vec{\mu} = (\mu_1, \ldots, \mu_{n+1}, \bar{\mu}_{n+1})$ and a $(n+2)\times(n+2)$ covariance matrix Σ specifying their variances and pairwise covariances. The only pdf f over these random variables that maximizes $h(f)$ while satisfying the provided means, variances, and covariances is the following function:

$$f(\vec{\theta}) = \frac{1}{\sqrt{(2\pi)^{n+2}|\Sigma|}} \exp\left(-\frac{1}{2}(\vec{\theta}-\vec{\mu})^T\Sigma^{-1}(\vec{\theta}-\vec{\mu})\right) \quad (2)$$

Later, in Section 3.4, we discuss why obtaining $\vec{\mu}$ and Σ is itself a challenge. However, even if we somehow obtain $\vec{\mu}$ and Σ, the next question is how to use pdf (2) to obtain an improved answer to a new query.

3.3 Improved Answer

In the previous section, we formalized the relationship among query answers, namely $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}, \bar{\boldsymbol{\theta}}_{n+1})$, as a joint pdf. In this section, we exploit this joint pdf to infer an improved answer to q_{n+1}. In other words, we need to find the most likely value for $\bar{\boldsymbol{\theta}}_{n+1}$ (the random variable representing q_{n+1}'s true answer), given the observed values for $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}$. Mathematically, INTELLI's improved answer $\hat{\theta}_{n+1}$ to q_{n+1} can be expressed as:

$$\hat{\theta}_{n+1} = \underset{\bar{\theta}_{n+1}}{\text{Arg Max}} \; f(\bar{\theta}_{n+1} \mid \boldsymbol{\theta}_1 = \theta_1, \ldots, \boldsymbol{\theta}_{n+1} = \theta_{n+1}) \quad (3)$$

Fortunately, a well-known result shows that conditionalizing the pdf (2) results in another normal distribution, which allows us to solve optimization (3) analytically [14]:

$$\hat{\theta}_{n+1} = \mu_{n+1} + k^T\Sigma_{sub}^{-1}(\vec{\theta}_{sub} - \mu_{n+1}), \quad (4)$$

where:

- k is a vector of length $n+1$ whose i-th element is $(i, n+2)$-th entry of Σ;
- Σ_{sub} is a $(n+1)\times(n+1)$ submatrix of Σ consisting of Σ's first $n+1$ rows and columns;
- $\vec{\theta}_{sub}$=$(\theta_1, \ldots, \theta_{n+1})^T$; and
- $\bar{\kappa}^2$ is the $(n+2, n+2)$-th entry of Σ.

Algorithm 1: The INFERENCE process

```
input  : μ,  Σ_s^{-1},  (q_n, θ_n, β_n)
output: (θ̂_{n+1}, β̂_{n+1}),  μ',  Σ'_s^{-1}
// Query Time Processing
θ̂_{n+1}, β̂_{n+1} ← Equation9 (μ, Σ_s^{-1}, θ_n, β_n)
display an improved answer (θ̂_{n+1}, β̂_{n+1})
// Post Processing:  Update mean and cov matrix
// Explained more in Section 4
μ' ← mean of (θ_i, ..., θ_{n+1})
Σ'_s^{-1} ← inverse of the covariance matrix of (θ_i, ..., θ_{n+1})
```

Likewise, the improved error $\hat{\beta}_{n+1}$ can be similarly obtained:

$$\hat{\beta}_{n+1}^2 = E[(\bar{\boldsymbol{\theta}}_{n+1} - \hat{\theta}_{n+1})^2] = \bar{\kappa}^2 - k^T\Sigma_{sub}^{-1}k \quad (5)$$

Note that, since the conditional pdf is a normal distribution, the *confidence interval*—the range that contains the true answer with a requested probability (e.g., 95%)—is computed in a straightforward manner.

In the next section, we discuss several key challenges involved in using these equations.

3.4 Key Challenges

As mentioned in Section 3.2, obtaining a joint pdf using (2) requires the knowledge of means, variances, and covariances of the random variables $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}, \bar{\boldsymbol{\theta}}_{n+1})$. However, acquiring these statistics is a non-trivial task. First, we have only observed one value for each of the random values $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}$, namely $\theta_1, \ldots, \theta_{n+1}$. Estimating variances and covariances of random variables from a single value is nearly impossible. Moreover, we do not have any observation for the last random variable, i.e., $\bar{\boldsymbol{\theta}}_{n+1}$. In Section 4.1, we present a solution to estimate these statistics by decomposing them into inter-tuple correlations in the underlying data. Then, in Section 4.2 we show how these individual correlations can be estimated indirectly and efficiently.

The second challenge is that estimating the improved answer $\hat{\theta}_{n+1}$ and error $\hat{\beta}_{n+1}$ using equations (4) and (5) will require matrix inversion of Σ_{sub}, which in general has an $O(n^3)$ time complexity. Since $\hat{\theta}_{n+1}$ and $\hat{\beta}_{n+1}$ need to be computed at query time, such a computation is infeasible in practice. We address this challenge in Section 5 by providing alternative mathematical expressions that are more amenable to efficient computation.

4. APPROXIMATE INFERENCE

As described in Section 3, that INTELLI expresses the relationship among query answers as a joint pdf over a set of random variables $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}, \bar{\boldsymbol{\theta}}_{n+1})$. In this process, we need to estimate the means, variances, and covariances of these random variables.

When performing inference with probabilistic distributions over random variables, the means of the random variables do not have a significant impact on the final results. This is because they only represent *a priori* knowledge of the data, and are hence updated in the process of conditioning the pdf [41].

Thus, in the rest of this section we focus on estimating the variances and covariances of these random variables. In Section 4.1, we propose a decomposition of the (co)variances of the random variables into *inter-tuple* covariance terms.

Then, in Section 4.2, we explain how inter-tuple covariances can be estimated analytically using parameterized covariance functions. In Section 4.3, we discuss the problem of determining optimal parameters for these covariance functions. Lastly, in Section 4.4, we present a safeguard against potential overfitting.

4.1 Covariance Decomposition

To compute the variances and covariances of our random variables (each representing possible answers to a query), we define and use *inter-tuple* covariances. This is based on the observation that the answer to a supported query can be mathematically represented as an integration over the attribute values of those tuples that satisfy the query's selection predicates.

Let g be an aggregate function on attribute A_k, $\boldsymbol{a}_k$ be a random variable representing the possible values of attribute A_k, $\boldsymbol{t} = (a_1, \ldots, a_l)$ be a vector of length l comprised of values for r's dimension attributes $A_1, \ldots, A_l$, and $\omega(\boldsymbol{t})$ be the fraction of tuples in r that have the same values for their dimension attributes as $\boldsymbol{t}$.

We use $\nu_g(\boldsymbol{t})$ to denote the expected value of aggregate function g given the values of dimension attributes. Thus, we have:

$$\nu_g(\boldsymbol{t}) = \begin{cases} E[\boldsymbol{a}_k \mid \boldsymbol{t}] & \text{if } g \text{ is } \texttt{AVG}(A_k) \\ \omega(\boldsymbol{t}) & \text{if } g \text{ is } \texttt{FREQ(*)} \end{cases}$$

Therefore, $\nu_g(\boldsymbol{t})$ is a random variable dependent on $\boldsymbol{t}$ (since relation r is drawn from an underlying distribution).

Let $\mathcal{F}_i$ denote the set of tuples satisfying the selection predicates of q_i. Then, the random variable $\bar{\boldsymbol{\theta}}_i$ representing possible true answers to q_i can be expressed in terms of $\nu_g(\boldsymbol{t})$:

$$\bar{\boldsymbol{\theta}}_i = \begin{cases} \dfrac{1}{\mathcal{Z}_i} \displaystyle\int_{\mathcal{F}_i} \nu_g(\boldsymbol{t})\, \omega(\boldsymbol{t})\, d\boldsymbol{t} & \text{if } g \text{ is } \texttt{AVG}(A_k) \\ \displaystyle\int_{\mathcal{F}_i} \nu_g(\boldsymbol{t})\, d\boldsymbol{t} & \text{if } g \text{ is } \texttt{FREQ(*)} \end{cases} \tag{6}$$

where $\mathcal{Z}_i$ is a normalization term defined as $\mathcal{Z}_i = \int_{\mathcal{F}_i} \omega(\boldsymbol{t})\, d\boldsymbol{t}$.

Based on equation (6), the covariance between random variables $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$ can be expressed as follows. (This quantity corresponds to the variance of $\boldsymbol{\theta}_i$ if $i = j$.)

$$\begin{aligned} E\left[(\boldsymbol{\theta}_i - \mu_i)(\boldsymbol{\theta}_j - \mu_j)\right] &= E\left[(\bar{\boldsymbol{\theta}}_i + \boldsymbol{\varepsilon}_i - \mu_i)(\bar{\boldsymbol{\theta}}_j + \boldsymbol{\varepsilon}_j - \mu_j)\right] \\ &= \begin{cases} \dfrac{1}{\mathcal{Z}_i \mathcal{Z}_j} \displaystyle\int_{\mathcal{F}_i} \int_{\mathcal{F}_j} \operatorname{cov}(\nu_g(\boldsymbol{t}), \nu_g(\boldsymbol{t}'))\, \omega(\boldsymbol{t}) \omega(\boldsymbol{t}')\, d\boldsymbol{t}'\, d\boldsymbol{t} \\ \qquad + \delta(i,j) \cdot \beta_i \qquad \text{if } g \text{ is } \texttt{AVG}(A_k) \\ \displaystyle\int_{\mathcal{F}_i} \int_{\mathcal{F}_j} \operatorname{cov}(\nu_g(\boldsymbol{t}), \nu_g(\boldsymbol{t}'))\, d\boldsymbol{t}'\, d\boldsymbol{t} \;+\; \delta(i,j) \cdot \beta_i \\ \qquad \text{if } g \text{ is } \texttt{FREQ(*)} \end{cases} \end{aligned} \tag{7}$$

where δ is the Kronecker delta function [41], and $\boldsymbol{\varepsilon}_i$ is a random variable with variance β_i representing the deviation of the raw answer from q_i's true answer.

To use equation (7), we must be able to compute the $\operatorname{cov}(\nu_g(\boldsymbol{t}), \nu_g(\boldsymbol{t}'))$ terms, which we call *inter-tuple covariances*. However, computing these inter-tuple covariances is challenging as we only have a single observation for each random variable $\nu_g(\boldsymbol{t})$. Moreover, even if we had a way to compute the inter-tuple covariance for arbitrary $\boldsymbol{t}$ and $\boldsymbol{t}'$, computing it for all possible pairs of tuples will be too costly and impractical. In the next section, we present an efficient alternative for estimating these inter-tuple covariances.

4.2 Efficient Computation of Covariances

To efficiently estimate the inter-tuple covariances, and thereby equation (7), we propose using *analytical covariance functions*, a well-known technique in statistical literature for approximating covariances [14]. In particular, INTELLI uses squared exponential covariance functions, which are proven to be *universal* [32], i.e., capable of approximating any target function arbitrary closely as the number of observations (here, query answers) increases. The squared exponential covariance function $\rho_g(\boldsymbol{t}, \boldsymbol{t}')$ is defined as:

$$\operatorname{cov}(\nu_g(\boldsymbol{t}), \nu_g(\boldsymbol{t}')) \approx \rho_g(\boldsymbol{t}, \boldsymbol{t}') = \sigma_g^2 \cdot \prod_{i=1}^{l} \exp\left(-\frac{\Delta(a_i, a_i')^2}{l_{g,i}^2}\right) \tag{8}$$

where

$$\Delta(a_i, a_i') = \begin{cases} |a_i - a_i'| & \text{if } A_i \text{ is numerical} \\ 1 - \delta(a_i, a_i') & \text{if } A_i \text{ is categorical} \end{cases}$$

Here, $\delta(a_k, a_k')$ is the Kronecker delta function [41] and $l_{g,i}$ for $i{=}1 \ldots m$ and σ_g^2 are tuning *parameters* to be learned from the underlying data.

Intuitively, when $\boldsymbol{t}$ and $\boldsymbol{t}'$ are similar, i.e., $\Delta(a_i, a_i')$ is small for most A_i, then $\rho_g(\boldsymbol{t}, \boldsymbol{t}')$ returns a larger value (closer to σ_g^2), indicating that the expected values of g for $\boldsymbol{t}$ and $\boldsymbol{t}'$ are highly correlated. Next, we describe how INTELLI automatically infers the most likely values of these tuning parameters.

4.3 Optimal Covariance Functions

To learn the tuning parameters of $\rho_g(\boldsymbol{t}, \boldsymbol{t}')$, INTELLI infers parameter values that are most likely to have generated the (observed) raw answers to past queries. Formally, INTELLI solves an optimization problem to find values for parameters $l_{g,1}, \ldots, l_{g,m}, \sigma_g^2$ that maximize the log-likelihood function:

$$\begin{aligned} &\log\, f(\vec{\theta}_{\text{past}}) = \\ &\quad -\frac{1}{2}(\vec{\theta}_{\text{past}} - \mu)^T \Sigma^{-1} (\vec{\theta}_{\text{past}} - \mu) - \frac{1}{2} \log|\Sigma| - \frac{n}{2} \log 2\pi \end{aligned}$$

where $f(\vec{\theta}_{\text{past}})$ is the joint pdf from Lemma 1, $\vec{\theta}_{\text{past}} = (\theta_1, \ldots, \theta_n)$, Σ is the $n \times n$ covariance matrix whose (i,j)-th entry is the (co)variance between $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$ computed using (8).

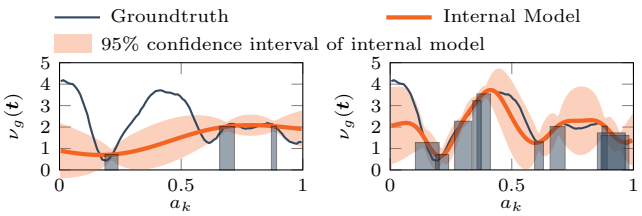
One caveat of using the above technique for choosing parameter values is the risk of overfitting, when the number of past observations (i.e., past query answers) is insufficient. We address this challenge next.

4.4 Overfitting Safeguard

Insufficient number of past queries may lead to parameter values that do not accurately reflect the characteristics of underlying data, causing INTELLI to underestimate its improved errors, as demonstrated in Figure 4. Thus, to detect and prevent overfitting, INTELLI relies on the following mechanism before returning the final answer $\hat{\theta}_{n+1}$ to the user:

$$\check{\theta}_{n+1} = \begin{cases} \hat{\theta}_{n+1} & \text{if } \theta_{n+1} - \beta_{n+1} \le \hat{\theta}_{n+1} \le \theta_{n+1} + \beta_{n+1} \\ \theta_{n+1} & \text{otherwise} \end{cases}$$

where $\hat{\theta}_{n+1}$ is the improved answer. In other words, INTELLI ignores its improved answer if it deviates too much from the



(a) After three queries **(b)** After ten queries

Figure 4: An example of how (a) an insufficient number of past queries can lead to overfitting (i.e., overly optimistic confidence intervals), and (b) how additional queries improve the model's accuracy. INTELLI relies on equation (4.4) to avoid overfitting.

raw answer. This mechanism guarantees that [36]:

$$E[(\check{\boldsymbol{\theta}}_{n+1} - \bar{\theta}_{n+1})^2] \leq 2\beta_{n+1}^2$$

where $\check{\boldsymbol{\theta}}_{n+1}$ is the random variable representing INTELLI's final answer and $\bar{\theta}_{n+1}$ is the true answer.

5. FORMAL GUARANTEES

The following theorem shows that INTELLI's improved answer is never worse than the raw answer.

Theorem 1. Let $(\theta_{n+1}, \beta_{n+1})$ be the raw answer and error returned by the AQP engine to a new query q_{n+1}, and $(\hat{\theta}_{n+1}, \hat{\beta}_{n+1})$. Then, at run-time, INTELLI can compute the improved answer and error $(\hat{\theta}_{n+1}, \hat{\beta}_{n+1})$ using the joint pdf of Lemma 1 in $O(n^2)$ time complexity, while guaranteeing the following:

$$\beta_{n+1} \geq \hat{\beta}_{n+1}$$

where the equality holds for β_{n+1}=0, i.e., when the AQP engine processes the entire data and produces an exact answer.

Proof. Since β_{n+1} and $\hat{\beta}_{n+1}$ are both non-negative, it suffices to show that $\beta_{n+1}^2 > \hat{\beta}_{n+1}^2$.

Let Σ be the covariance matrix of the vector $\vec{\boldsymbol{\theta}}$=($\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n+1}, \bar{\boldsymbol{\theta}}_{n+2}$), k_s be a column vector of length n whose i-th element is the $(i, n+1)$-th entry of Σ, Σ_s be an $n \times n$ submatrix of Σ that consists of Σ's first n rows/columns, $\bar{\kappa}^2$ be a scalar value at the $(n+2, n+2)$-th entry of Σ, and $\vec{\theta}_s$ be a column vector $(\theta_1, \ldots, \theta_n)^T$.

We can express k and Σ_{sub} of equation (4) in block forms as follows:

$$k = \begin{pmatrix} k_s \\ \bar{\kappa}^2 \end{pmatrix}, \quad \Sigma_{sub} = \begin{pmatrix} \Sigma_s & k_s \\ k_s^T & \bar{\kappa}^2 + \beta_{n+1}^2 \end{pmatrix}, \quad \vec{\theta}_{sub} = \begin{pmatrix} \vec{\theta}_s \\ \theta_{n+1} \end{pmatrix}$$

Using the matrix inversion in block form [28], Σ_{sub}^{-1} can also be expressed as:

$$\Sigma_{sub}^{-1} = \begin{pmatrix} \Sigma_s^{-1} + \frac{1}{\gamma^2}\Sigma_s^{-1}k_s k_s^T \Sigma_s^{-1} & -\frac{1}{\gamma^2}\Sigma_s^{-1}k_s \\ -\frac{1}{\gamma^2}k_s^T\Sigma_s^{-1} & \frac{1}{\gamma^2} \end{pmatrix}$$

where $\gamma^2 = \bar{\kappa}^2 - k_s^T \Sigma_s^{-1} k_s$.

Plugging the above block inverse form into equation (4) and (5) reduces, after some simplification, to the following expressions:

$$\gamma^2 = \bar{\kappa}^2 - k_s^T \Sigma_s^{-1} k_s$$

$$\hat{\theta}_{n+1} = \mu_{n+1} + \frac{\beta_{n+1}^2 \cdot k_s^T \Sigma_s^{-1}(\vec{\theta}_s - \mu_{n+1}) + \gamma^2 \cdot (\theta_{n+1} - \mu_{n+1})}{\gamma^2 + \beta_{n+1}^2}$$

$$\hat{\beta}_{n+1}^2 = \gamma^2 \cdot \beta_{n+1}^2 / (\gamma^2 + \beta_{n+1}^2) \tag{9}$$

Note that expression (9) involves a matrix inversion Σ_s^{-1}, but since Σ_s only contains the (co)variances of pairs of past query answers, INTELLI can pre-compute Σ_s.

To derive the relationship between $\hat{\beta}_{n+1}^2$ and β_{n+1}^2, we subtract $\hat{\beta}_{n+1}^2$ from β_{n+1}^2:

$$\beta_{n+1}^2 - \frac{\gamma^2 \cdot \beta_{n+1}^2}{\gamma^2 + \beta^2} = \beta_{n+1}^2 \left(1 - \frac{\gamma^2}{\gamma^2 + \beta_{n+1}^2}\right)$$

This expression is equal to 0 if $\beta_{n+1} = 0$, and is otherwise larger than 0. Thus, the theorem holds. □

6. DATA UPDATES

INTELLI supports tuple insertions.[9] A naïve strategy would be to re-execute all past queries every time new tuples are added to the database to obtain their updated answers. This solution is obviously impractical.

Instead, here we show how INTELLI can still make use of answers to previous queries even when new tuples have been added since computing those answers. The basic idea is to simply lower our confidence in the raw answers of past queries.

Assume that q_i (whose aggregate function is on A_k) is computed on an old relation r, and a set of new tuples r' has since been added to r to form an updated relation r^u. Let $\bar{\boldsymbol{\theta}}_i'$ be a random variable representing the possible values for q_i's true answer on r', and $\bar{\theta}_i^u$ be q_i's true answer on r^u.

We represent the possible difference between A_k's values in r and those in r' by a random variable $\boldsymbol{s}_k$ with mean α_k and variance η_k^2. Thus:

$$\bar{\boldsymbol{\theta}}_i' = \bar{\theta}_i + \boldsymbol{s}_k$$

The values of α_k and η_k^2 are estimated using small samples of r and r'. INTELLI uses the following lemma to update the raw answer and raw error for q_i (see [36] for proof).

Lemma 2.

$$E[\bar{\theta}_i^u - \boldsymbol{\theta}_i] = \alpha_s \cdot \frac{|r'|}{|r| + |r'|}$$

$$E[(\bar{\theta}_i^u - \boldsymbol{\theta}_i - \alpha_s \cdot \frac{|r'|}{|r| + |r'|})^2] = \beta_i^2 + \left(\frac{|r'|}{|r| + |r'|} \cdot \eta_k\right)^2$$

where $|r|$ and $|r'|$ are the number of tuples in r and r', respectively.

Once the raw answers and the raw errors of past queries are updated using this lemma, the remaining inference process remains the same.

[9] Other forms of data updates (e.g., deletion) are not supported, as INTELLI is currently implemented atop SparkSQL which (similar to other HDFS-based engines) is an append-only database.

Dataset	# Analyzed	# Supported	Percentage
VerticaLog	3,342	2,463	73.7%
TPC-H	21	14	63.6%

Table 3: Generality of Intelli. Intelli supports a large fraction of real-world and benchmark queries.

7. EXPERIMENTS

Our experiments aim to (1) quantify the percentage of real-world queries that benefit from Intelli and their average speedup (Sections 7.2 and 7.3), (2) vet the reliability of Intelli's error estimates (Section 7.4), (3) measure Intelli's computational overhead and memory footprint (Section 7.5), (4) study the impact of different workloads and data distributions on Intelli's effectiveness (Section 7.6), and (5) evaluate Intelli's ability in coping with changing datasets (Section 7.7).

In summary, our results indicate the following:

- Intelli supports a large fraction (73.7%) of aggregate queries in a real-world workload, bringing significant speedups (upto 10.43x) compared to existing (sampling-based) AQP solutions.
- Given the same processing time, Intelli reduces the baseline's approximation error on average by 56–95%.
- Intelli's run-time overhead is <10 miliseconds (0.12–0.52%) and its memory footprint is negligible.
- Intelli's approach is robust against different workloads and data distributions.

7.1 Experiment Setup

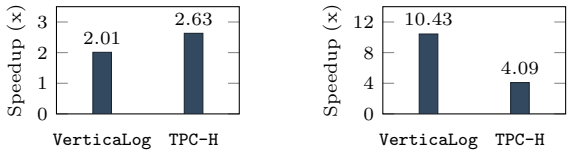
Datasets and Query Workloads — For our experiments, we used the three datasets described below:

1. `VerticaLog`: This is a real-world query trace from one of HP Vertica's largest customers (anonymized). This dataset contains 310 tables and 15.5K timestamped queries issued between March 2011 and April 2012, 3.3K of which queries are analytical queries supported by SparkSQL. We did not have the customer's original dataset but had access to their data distribution, which we used to generate a 536GB dataset.
2. `TPC-H`: This is a well-known analytical benchmark with 22 query types, 21 of which contain at least one aggregate function (including 2 queries with `min` or `max`). We used a scale factor of 100, i.e., the total data size was 100GB. We generated a total of 500 queries using `TPC-H`'s workload generator using default settings. using `TPC-H`'s query generator.
3. `Synthetic`: For more controlled experiments, we also generated large-scale synthetic datasets with different distributions (see Section 7.6 for details).

Implementation — For comparative analysis, we implemented two systems on top of SparkSQL [12], a relational engine shipped with Spark (ver 1.5.1):

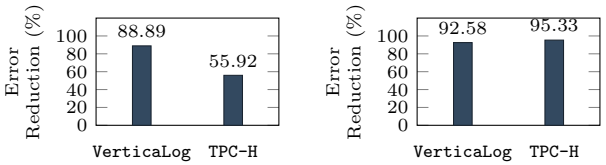
1. NoLearn: This system runs queries on *samples* of the original tables to obtain fast but approximate query answers and their associated statistical errors. This is the same approach taken by existing AQP engines, such as [4, 5, 9, 16, 40, 42, 49].
2. Intelli: This system invokes NoLearn to obtain raw answers/errors but modifies them to produce improved answers/errors using our proposed inference process.

Experimental Environment — We used a Spark cluster (for both NoLearn and Intelli) using 5 Amazon EC2 `m4.2xlarge` instances, each with 2.4 GHz Intel Xeon E5 processors (8 cores) and 32GB of memory. Our cluster also included SSD-backed HDFS [44] for Spark's data loading. For experiments with cached datasets, we distributed Spark's RDDs evenly across the nodes using SparkSQL DataFrame `repartition` function.



(a) Fully Cached **(b)** No Cache

Figure 5: Speedup of Intelli over NoLearn for a target error of 2%.



(a) Fully Cached **(b)** No Cache

Figure 6: Average error reduction of queries by Intelli (compared to NoLearn) for the same time budget.

7.2 Generality of Intelli

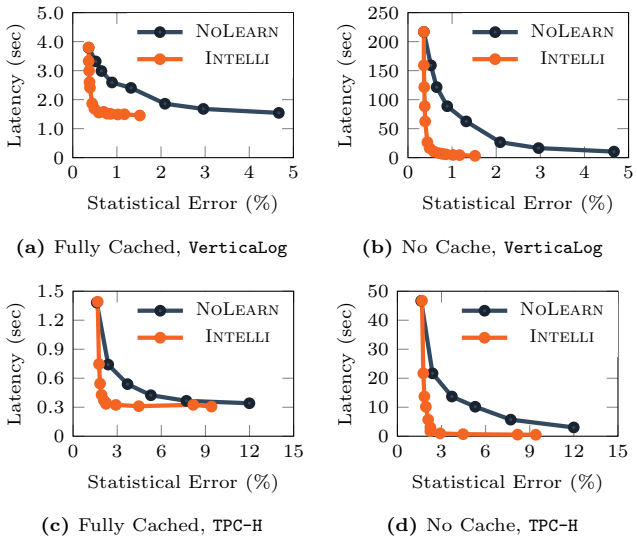
To quantify the generality of our approach, we analyzed the real-world SQL queries in `VerticaLog`. From the original 15.5K queries, SparkSQL was only able to process 3.3K of the aggregate queries. Among those 3.3K queries, Intelli supported 2.4K queries, i.e., 73.9% of the analytical queries could benefit from Intelli. In addition, we analyzed the 21 `TPC-H` queries and found 14 queries supported by Intelli. Others could not be supported due to textual filters or disjunctions in the `where` clause. These statistics are summarized in Table 3. This analysis proves that Intelli can support a large class of analytical queries in practice. Next, we quantify how much these supported queries benefit from Intelli.

7.3 Query Speedup

In this section, we study Intelli's query speedup over NoLearn: how faster can queries achieve the same level of accuracy under Intelli compared to NoLearn? For this experiment, we used each of `VerticaLog` and `TPC-H` datasets in two different settings. In one setting, all samples were cached in the memories of the cluster, while in the second, SparkSQL had to read the data from SSD-backed HDFS.

We allowed both systems to process half of the queries (since `VerticaLog` queries were timestamped, we used the first half). While processing those queries, NoLearn simply returned the query answers but Intelli also learned its model parameters. Then, for the second half of the queries, we recorded both systems' query response times (i.e., latencies), approximate query answers, and statistical errors.

Figure 5 shows the average latencies of the two systems for different datasets and caching scenarios, for a target error of 2%. For `VerticaLog` queries on non-cached data, Intelli delivered a 10.43x speedup on average (i.e., more than 90%



(a) Fully Cached, `VerticaLog` **(b)** No Cache, `VerticaLog`

(c) Fully Cached, `TPC-H` **(d)** No Cache, `TPC-H`

Figure 7: The trade-off between statistical error and latency for NoLearn and Intelli.

reduction in query response time) for the same target error. This is because Intelli could deliver highly-accurate answers using significantly smaller sample sizes by leveraging the inferred knowledge from past query answers. In all cases, Intelli achieved at least 2x speedup over NoLearn.

Figure 6 shows the average error reduction of Intelli compared to NoLearn, for a fixed time budget (i.e., target latency). In this experiment, the target latencies were 1.5 sec for cached `VerticaLog`, 3.5 sec for non-cached `VerticaLog`, and 0.5 sec for both cached and non-cached `TPC-H`. (Other target latencies will be reported in Figure 7.) As shown in Figure 6, for the same time budget, Intelli reduced NoLearn's error by at least 55.92% and up to 95.33%.

Figure 7 presents a detailed study of the trade-off between average query latencies and average statistical errors in both systems. In all experiments, the latency-error graphs exhibit consistent patterns: (1) Intelli produced smaller statistical errors even when target latencies were very small, and (2) Intelli showed faster query response times for the same target statistical errors. Due to the asymptotic nature of statistical errors, achieving extremely accurate answers (e.g., less than 0.5%) requires relatively large sample sizes (and processing times) even for Intelli. Also, the reason that Intelli's speedups were slightly lower for cached settings was that the default overhead of SparkSQL was relatively large compared to its overall data processing time due to the sample size. In other words, even if Intelli reduced the query processing time to zero, it would not be possible to achieve more than 3–5x speedups due to SparkSQL's overhead of running an empty query. However, Intelli still achieved 2.63x speedup even for fully-cached datasets.

7.4 Reliability of Statistical Error Guarantees

To confirm the validity of Intelli's probabilistic error guarantees, we configured Intelli to run each query with different statistical error bounds at 95% confidence and measured the actual error in each case (see Section 3.3 for details). Figure 8 shows the 5% percentile, mean, and 95% percentile of the actual errors across different queries. The

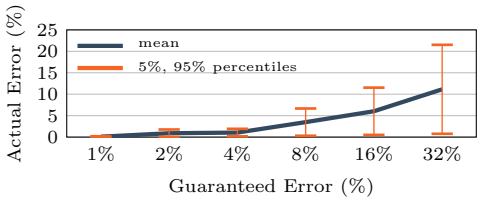


Figure 8: Correctness of Intelli's statistical error guarantees.

Latency	Cached	No-Cache
NoLearn	2.083 sec	52.50 sec
Intelli	2.093 sec	52.51 sec
Overhead	0.010 sec (0.48%)	0.010 sec (0.02%)

Table 4: The runtime overhead of Intelli.

results demonstrate that Intelli's statistical error guarantees were almost never violated, thanks to its overfitting safeguard mechanism.

7.5 Memory and Computational Overhead

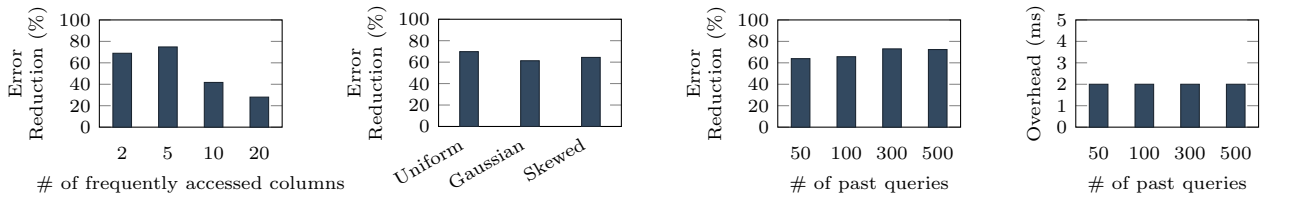
In this section, we study Intelli's additional memory footprint (due to query synopsis) and its runtime overhead (due to inference). The total size of the generated snippets was on average 150KB per query for `VerticaLog`, and 8 KB per query for `TPC-H`. This is because Intelli only stores the first N^{max}=1000 tuples from each query's answer and does not keep any of the input tuples.

To measure Intelli's runtime overhead, we recorded the time spent for its regular query processing (the same component as NoLearn) and the additional time spent for the inference and updating the final answer. As summarized in Table 4, the runtime overhead of Intelli was negligible compared to the overall query processing time. This is because multiplying a vector by a $C_g \times C_g$ matrix does not take much time compared to regular query planning, processing, and network commutations among the distributed nodes. (Note that C_g=100 by default; see Section 2.4.)

7.6 Impact of Different Data Distributions and Workload Characteristics

In previous sections, we reported experiments on two datasets (`VerticaLog` and `TPC-H`). In this section, we generated various synthetic data and queries to fully understand how Intelli's effectiveness changes for different data distributions, query patterns, and number of past queries.

First, we studied the impact of having queries with a more diverse set of columns in their selection predicates. Thus, we produced a table of 50 columns and 5M rows, and generated four different query workloads with varying number of frequently accessed columns. The columns used for the selection predicates were chosen according to a power-law distribution. Specifically, a fixed number of columns (called *frequently access columns*) had the same probability of being accessed, but the access probability of the remaining columns decayed according to the power-law distribution. For instance, if the number of frequently access columns was 10, the first ten columns appeared with equal probability in each query, but the probability of appearance reduced by half for every remaining column. Figure 9(a) shows that as the number of frequently accessed columns increased, Intelli's relative error reduction over NoLearn gradually decreased (the number of past queries were fixed to 100). This



(a) Workload Diversity **(b)** Data distribution **(c)** Learning Behavior **(d)** Overhead Consistency

Figure 9: The effectiveness of INTELLI in reducing NOLEARN's error for different (a) levels of diversity in the queried columns, (b) data distributions, and (c) number of past queries observed. Figure (d) shows INTELLI's overhead for different number of past queries.

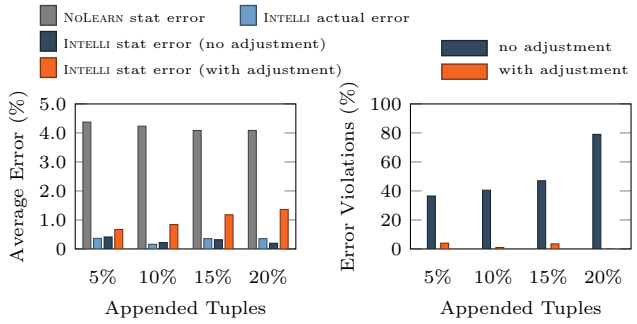


Figure 10: The Section 6's adjustment technique is highly effective in delivering correct error estimates in face of new data.

is expected as INTELLI constructs its internal model based on the columns appearing in the past. In other words, to cope with the more diversity, more past queries are needed to understand the complex underlying distribution generating the data. Note that, according to the analytic queries in the `VerticaLog` dataset, most of the queries included less than 5 distinct selection predicates. However, by processing more queries, INTELLI continues to learn more about the underlying distribution, producing larger error reductions even when the workload is extremely diverse.

Second, to study INTELLI's potential sensitivity to we generated three tables with three different probability distributions: uniform, Gaussian, and a log-normal (skewed) distribution. Figure 9(b) shows INTELLI's error reductions when queries were run against each table. INTELLI delivered a consistent performance regardless of the underlying distribution. This is due to the power and generality of the maximum entropy principle taken by INTELLI.

Third, we varied the number of past queries observed by INTELLI before running our test queries. Figure 9(c) demonstrates that the error reduction keeps increasing until seeing 300 queries and then leveled off. This is due to the asymptotic nature of statistical errors, enabling INTELLI to deliver reasonable performance without having to observe too many queries.

Lastly, we studied the negative impact of increasing the number of past queries on INTELLI's overhead. Since INTELLI's inference consists of a small matrix multiplication we did not observe a noticeable increase as the number of queries in the query synopsis increased (Figure 9(d)).

7.7 Supporting Data Appends

In this section, we study the impact of new data (i.e., tuple insertions) on INTELLI's effectiveness. Similar to the previous section, we generated an initial synthetic table with 5M tuples and appended additional tuples to generate different versions of the table. The newly inserted tuples were generated such that their attribute values gradually diverged from the attribute values of the original table. We distinguish between these different versions by the ratio of their newly inserted tuples, e.g., a 5% appended table means that 250K (5% of 5M) tuples were added. We then ran the queries and recorded the average statistical errors of INTELLIADJUST and INTELLINOADJUST (our approach with and without the technique introduced in Section 6). We also measured the statistical error of NOLEARN and the actual error.

As shown in Figure 10(a), INTELLINOADJUST produced overly-optimistic statistical errors (i.e., lower than the actual error) for 15% and 20% appends, whereas INTELLIADJUST produced valid statistical errors in all cases. Since this figure shows the *average* statistical errors across all queries, we also computed the fraction of individual queries for which each method's statistical error was violated. In Figure 10(b), the Y-axis indicates those cases where the actual error was higher than the guaranteed statistical error. This figure shows more error violations for INTELLINOADJUST, which increased with the number of new tuples. In contrast, INTELLIADJUST produced valid statistical errors in most cases, while delivering substantial error reductions compared to NOLEARN.

8. RELATED WORK

Approximate Query Processing — There has been substantial work on sampling-based approximate query processing [6, 7, 9, 13, 16, 21, 34, 45]. Some of these systems differ in their sample generation strategies. For instance, STRAT [16] and AQUA [7] create a single stratified sample, while BlinkDB creates samples based on *column sets*. Online Aggregation (OLA) [17,24,35,47] continuously refine its answers during query execution. Other have focused on obtaining faster or more reliable error estimates [8,48]. These are orthogonal to our work, as reliable error estimates from an underlying AQP engine will also benefit DBL.

Adaptive Indexing, View Selection — Adaptive Indexing and database cracking [25, 27, 39] incrementally update indices as part of query processing in order to speed up future queries accessing previously accessed tuples. Materialized views are another means of speeding up future queries [20, 23, 30]. While these techniques are orthogonal to DBL (i.e., they can be used in the underlying AQP engine), they are fundamentally different than DBL (refer to Previous Approaches in Section 1).

Pre-computation — COSMOS [47] stores the (exact) results of past queries as multi-dimensional cubes. These aggregated cubes are then re-used if they are contained in the

new query's input range, while boundary tuples are read from the database. This approach is not probabilistic and is limited to low-dimensional data due to the exponential explosion in the number of possible cubes. Also, similar to view selection, COSMOS relies on strict query containment.

Model-based and Statistical Databases — Statistical approaches have been used in databases for various goals. For example, MauveDB [18] constructs views that express a statistical model, hiding the possible irregularities of the underlying data. MauveDB's goal is to support statistical modeling, such as regression or interpolation, rather than speeding up future queries. BayesDB [31] provides a SQL-like language that enables non-statisticians to declaratively use various statistical models.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented database learning, a novel approach to exploit past queries' (approximate) answers in speeding up new queries using a principled statistical methodology. We presented a prototype of this vision, called INTELLI, on top of SparkSQL. Through extensive experiments on real-world and benchmarks query logs, we demonstrated that INTELLI supports 73.7% of real-world analytical queries, speeding them up by up to 10.43x compared to existing sampling-based approximation engines.

Exciting lines of future work include: (1) study other inferential techniques for realizing database learning, (2) develop an idea of *active database learning*, whereby the engine itself proactively executes certain approximate queries that can best improve its internal model, and (3) extend INTELLI to support visual analytics [37].

10. REFERENCES

[1] https://db.apache.org/derby/docs/10.6/tuning/ctuntransform36368.html.

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

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

[4] SnappyData. http://www.snappydata.io/.

[5] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *VLDB*, 1999.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[12] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.

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

[14] C. M. Bishop. Pattern recognition. *Machine Learning*, 2006.

[15] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An overview of machine learning. In *Machine learning*. 1983.

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

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

[18] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD*, 2006.

[19] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2009.

[20] A. El-Helw, I. F. Ilyas, and C. Zuzarte. Statadvisor: Recommending statistical views. *VLDB*, 2009.

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

[22] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *ICDEW*, 2010.

[23] A. Y. Halevy. Answering queries using views: A survey. *VLDBJ*, 2001.

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

[25] S. Idreos. Database cracking: Towards auto-tuning database kernels. *CWI and University of Amsterdam*, 2010.

[26] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[27] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[28] J. S. R. Jang. General formula: Matrix inversion lemma. http://www.cs.nthu.edu.tw/~jang/book/addenda/matinv/matinv/.

[29] Y. Jia. Running tpc-h queries on hive. https://issues.apache.org/jira/browse/HIVE-600.

[30] S. Joshi and C. Jermaine. Materialized sample views for database approximation. *TKDE*, 2008.

[31] V. Mansinghka et al. Bayesdb: A probabilistic programming system for querying the probable implications of data. *arXiv*, 2015.

[32] C. A. Micchelli, Y. Xu, and H. Zhang. Universal kernels. *JMLR*, 2006.

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

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

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

[36] Y. Park. Technical report for database learning. http://web.eecs.umich.edu/~pyongjoo/dblt.pdf.

[37] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. *arXiv*, 2015.

[38] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, 2015.

[39] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, 2015.

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

[41] C. E. Rasmussen. Gaussian processes for machine learning. 2006.

[42] F. Rusu, C. Qin, and M. Torres. Scalable analytics model calibration with online aggregation. *IEEE Data Eng. Bull.*, 2015.

[43] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 1959.

[44] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

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

[46] J. Skilling. *Data Analysis: A Bayesian Tutorial*. Oxford University Press, 2006.

[47] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, 2010.

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

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