DAQ: A New Paradigm for Approximate Query Processing

Navneet Potti
University of Wisconsin – Madison
nav@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin – Madison
jignesh@cs.wisc.edu

ABSTRACT

Many modern applications deal with exponentially increasing data volumes and aid business-critical decisions in near real-time. Particularly in exploratory data analysis, the focus is on interactive querying and some degree of error in estimated results is tolerable. A common response to this challenge is approximate query processing, where the user is presented with a quick confidence interval estimate based on a sample of the data. In this work, we highlight some of the problems that are associated with this *probabilistic* approach when extended to more complex queries, both in semantic interpretation and the lack of a formal algebra. As an alternative, we propose *deterministic* approximate querying (DAQ) schemes, formalize a closed deterministic approximation algebra, and outline some design principles for DAQ schemes. We also illustrate the utility of this approach with an example deterministic online approximation scheme which uses a bitsliced index representation and computes the most significant bits of the result first. Our prototype scheme delivers speedups over exact aggregation and predicate evaluation, and outperforms sampling-based schemes for extreme value aggregations.

1 Introduction

As organizations collect ever-larger volumes of data and use analytics to drive their decision-making processes, the focus is often less on exactness of the result and more on timeliness or responsiveness, particularly for interactive exploratory analysis. This need is often best met with quick and approximate estimates that are either within a user-specified error tolerance or are continuously updated to be more exact over time. The most common querying paradigm in this direction is *sampling-based* approximate querying (which we call SAQ) [1, 3], where the computation is performed over a small random subset of the data. A special case is *online aggregation* [6], where a running estimate based on data seen thus far is continuously updated as the computation proceeds. The error in the estimate is specified using a confidence interval or error bars.

Proceedings of the VLDB Endowment, Vol. 8, No. 9

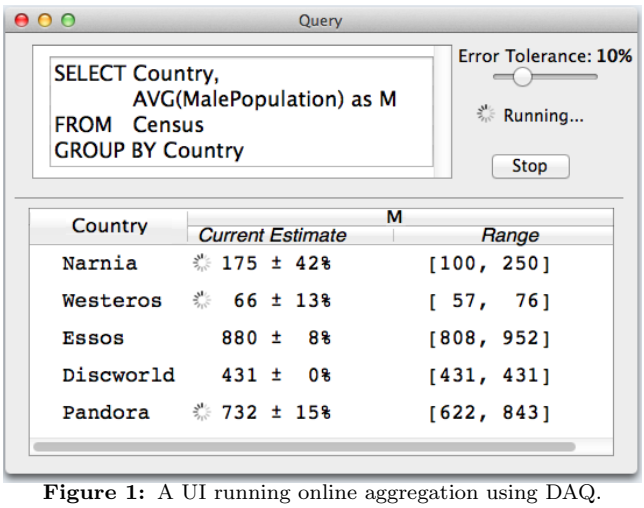


Figure 1: A UI running online aggregation using DAQ.

SAQ schemes suffer from various shortcomings. The confidence interval estimation does not lend itself to a formal closed algebra, making it harder to formally reason about complex queries. Sometimes this method is coupled with bootstrapping [12] and the query is run on multiple artificial samples obtained by resampling the original sample. But this approach can be computationally expensive and inaccurate for some queries [2]. The confidence intervals also ignore the tails of the data distributions entirely, making it harder to use them when we are interested in the extreme values (e.g. aggregations like `MAX`). It can also be difficult for lay users to correctly interpret the semantics of confidence intervals, particularly when presented with a large number of independent estimates. (Section 2 has more details.)

In this work, we propose querying paradigm called *deterministic* approximate querying, abbreviated as DAQ. We define the DAQ approach informally for now as one where the precision of the estimate is maintained and presented in terms of deterministically guaranteed lower and upper bounds. The user interface in Figure 1 illustrates this concept using a screenshot of the intermediate results of a query in progress. The example query is an aggregation on the `Census` table excerpted in Table 1. The aggregate requested by the user is displayed using an interval (under "`Range`") which is deterministically guaranteed to include the true average. For instance, the average `MalePopulation` for `Narnia` is guaranteed to be between 100 and 150, and the 42% "error" in the estimate is the ratio of the width of the interval relative to its midpoint. As the aggregation proceeds, the intervals get narrower and the error decreases. This process

Country	City	Male Population	Female Population
Far Away Galaxy	Alderaan	1,001,453,234	1,000,671,173
Far Away Galaxy	Coruscant	12,876,461,144	13,750,815,931
Far Away Galaxy	Naboo	297,540,057	301,810,369
Far Away Galaxy	Tatooine	103,488	97,001
Far Away Galaxy	Kashyyyk	22,541,803	23,177,900
Middle-Earth	Shire	1,654,784	1,432,605
Middle-Earth	Rivendell	76,236	76,902
Middle-Earth	Minas Morgul	30,204,645	25,784,115
Middle-Earth	Minas Tirith	15,721,104	15,438,168

Table 1: Example tuples from a large `Census` table.

continues untill either all the errors are within the tolerance specified by the user, or the user terminates the session.

These deterministic guarantees remedy the issues that are associated with SAQ schemes highlighted above by simplifying the semantics, making them easier to reason about and interpret correctly. This approach allows us to develop a formal closed algebra for such schemes in Section 3 so that the approximation schemes are applicable to arbitrarily complex queries. We mention some of the desired properties of such schemes and outline some example schemes as well. We delve into one such deterministic online approximation scheme, the *Bitwise approximation scheme*, in Section 5. In our prototype, termed Bitsliced Index implementation, predicate evaluation and aggregation are performed using a bitsliced index representation of the columns [10, 14], and the query result is computed in order of significance of the digits in the result. The empirical data presented in Section 6 shows that this scheme allows efficient approximate evaluation of queries, particularly when computing aggregates that depend on extreme values in heavy-tailed distributions.

Overall, our contributions lie in proposing the DAQ approach, developing a robust theoretical framework for handling complex queries, and empirically demonstrating the benefits and limitations of our methods. In our initial implementation, we have limited our scope to a restricted class of queries and focused on read-mostly environments. To the best of our knowledge this is the first proposal for deterministic approximate query processing, and as discussed in Section 8, there are many interesting directions for future work. We also acknowledge that this paper does not close the chapter on DAQ, but rather is its opening chapter.

We also note that DAQ and SAQ schemes are complementary to each other, as they use distinctly different techniques and provide semantically different error guarantees. While this introductory work delves into how these approaches *differ* from each other, these approaches can certainly also be *combined*, and we believe that looking at combining both schemes presents a promising avenue for future work.

2 The need for DAQ over SAQ

We begin with an example to motivate the DAQ paradigm. Consider a large `Census` relation such as the one excerpted in Table 1 containing the populations of a large number of cities and countries, broken out by gender. We will use this relation as a running example for the rest of this paper.

Suppose that we want to find the maximum `MalePopulation` across any `City` in this relation. The SQL query is:

```
SELECT MAX(MalePopulation) FROM Census
```

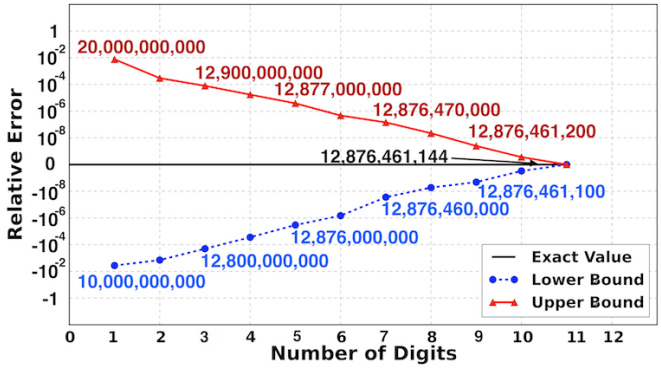


Figure 2: Relative errors in the lower and upper bounds when evaluating `MAX(MalePopulation)` in `Census`.

In SAQ schemes, this `MAX` value is estimated by evaluating the query on a small random sample of the relation. Naturally, aggregation functions such as `MAX` that are sensitive to outliers (i.e. rare, extreme values) are difficult to approximate well, as they are unlikely to occur in the sample.

In DAQ schemes, we find an interval within which the maximum value is guaranteed to lie, and then iteratively refine this interval by tightening the lower and upper bounds. For instance, suppose that we examine the values in the `MalePopulation` column one digit at a time. Then, a simple approximation scheme would proceed by examining these digits in the order of the most significant digits first and update the estimate in the same order. This approach is illustrated in Figure 2, where the two curves represent our lower and upper bound estimates in successive iterations of this algorithm. On the x-axis is the number of iterations, which is also the number of digits we have examined from each row of the column. The relative error on the y-axis is computed as the ratio of the error in our estimate to the exact maximum. After examining just the first digit in each row for the entire column, we find that the maximum is around 10 billion. Since we have ignored the remaining 10 digits, we can claim that the error in our estimate is between 0 and (10 billion - 1). So we use the interval [10 billion, 20 billion] to represent our estimate. Subsequently, as we examine more digits, we add more significant digits to our estimate and tighten our bounds, as shown by the curves. In fact, the relative error in our estimate falls exponentially till we reach the exact result (after examining all 11 digits).

The discussion above provides an informal view of our DAQ approach, leaving the theoretical and implementation aspects to subsequent sections. Next, we further motivate the need for, and the advantages of, our approach.

2.1 Where SAQ Falls Short

The SAQ approach provide approximate results for a query by running it on small random samples of an "appropriate" size. The error in the estimate is maintained and presented in the form of confidence intervals.

Suppose that we are interested in the average of the attribute `MalePopulation` across all cities, i.e. the query is:

```
SELECT AVG(MalePopulation) FROM Census
```

Depending on whether the city `Coruscant` is excluded or included in the sample, the estimated average for the above dataset can vary from 171 million to 1.58 billion. In the former case, let us assume the average is estimated with 95% confidence as 171 million ± 100 million. This interval

indicates that the true average for the entire dataset is between 71 million and 271 million with 95% probability. The large width of the interval reflects the high variance in the column. Note that this probabilistic estimate gives us no information about how far off the exact average might be in the tails (which occur with 5% probability). Furthermore, consider a query such as the following:

```
SELECT Country, AVG(MalePopulation)
FROM Census GROUP BY Country
```

If there are a large number of tuples in the result, each with a presumably reliable 95% confidence interval for the average `MalePopulation` in the corresponding `Country`, then we are nearly guaranteed that in 5% of these cases, the exact average will exceed the estimated interval. SAQ methods make it hard to correctly reason about the semantics of the probabilistic estimates.

This situation is exacerbated when we seek aggregates that reflect extreme values, such as `MAX`, `MIN` or `TOP k`. The SAQ schemes either provide weak guarantees for the estimates or require large sample sizes.

The concerns highlighted above relate to the complex semantics of the confidence interval, and its inability to capture extreme values. Next, consider a more complex query that highlights a separate theoretical challenge.

```
SELECT Country FROM
  ( SELECT Country,
      AVG(MalePopulation) AS M,
      AVG(FemalePopulation) AS F
    FROM Census GROUP BY Country
  ) WHERE M > F
```

Here, we would like to find the countries where the average `MalePopulation` exceeds the average `FemalePopulation`. Ideally, we want to reuse the approach and results of the previous query, first finding confidence intervals that estimate the averages (`M` and `F`), then comparing them (`M > F`). Such a closed algebra becomes particularly appealing when the subquery is complex (e.g with many levels of nesting). However, the confidence interval representation has no clear semantics for the comparison operation: is the probabilistic estimate 100 ± 10 "greater than" 90 ± 20? In a closed algebra, we want to associate an *inclusion probability* that the corresponding `Country` satisfies this predicate, as in [4]. However, the representation of approximation errors using confidence intervals alone is not sufficient for this purpose. We run into similar problems for queries such as one that computes the average gender imbalance (`M - F`).

Note that complex queries (though not extreme value aggregates) can be handled in SAQ schemes using bootstrapping, but this approach is often expensive. As we show next, the DAQ schemes lead to a more natural approach where we define a formal closed algebra that is theoretically capable of handling arbitrarily complex relational queries.

2.2 How DAQ Helps

The key idea in the DAQ paradigm is to eschew the probabilistic guarantees provided by SAQ in favor of deterministic guarantees. To represent an approximation to an attribute a, rather than using a confidence interval, we use a deterministic interval bounding it: $l_{\bar{a}} \leq a \leq u_{\bar{a}}$. To obtain a closed algebra, we also extend this notion to relations: a relation R is approximated using a pair of relations (sets of tuples) $L_{\bar{R}} \subseteq R \subseteq U_{\bar{R}}$. Intuitively, these relations bound the membership of tuples in the exact relation R. Every tuple that is *potentially* in the exact relation R is guaranteed to be in our upper bound $U_{\bar{R}}$ and every tuple that is in our lower bound $L_{\bar{R}}$ is *certainly* in the exact relation R. These bounds allow us to perform approximate evaluation of complex queries, since every intermediate attribute and relation has the same representation: a generalization of the intervals we used in Figure 2.

This representation and resulting closed algebra are explained more formally in Section 3. The general approach is to begin with an interval that is deterministically guaranteed to include the exact value (whether an attribute resulting from an aggregation, or a relation resulting from a selection with a predicate), then iteratively tighten the intervals either until the estimates are within the desired tolerance for approximation error, or until the estimate becomes exact (i.e. both the lower and upper bounds become equal).

Since the intervals are deterministic bounds, we avoid the problem of estimating tail values. For efficiency, the DAQ schemes must be designed to compute the most significant part of the result first. For instance, in the example shown in Figure 2, we did so by computing the result in the order of the most significant digit first. Such a strategy allows us to bound the uncertainty in any intermediate result using bounds on the value of the remaining (least significant) digits. As the computation proceeds, in each iteration, we tighten these bounds by including more digits in the aggregation, leaving fewer uncertain digits of smaller significance. This approach naturally leads to fast approximation for aggregation functions that are sensitive to the extreme values, such as `MAX` and `TOP k`. This approach can also be extended to comparisons using interval algebra and 3-valued logic. For brevity, we have excluded the extension to arithmetic operations, which uses interval arithmetic.

3 Formal Definition of DAQ

In this section, we formally define the DAQ framework and develop its algebra, using the notation shown in Table 2.

3.1 Attributes and Relations

We use a generalized notion of intervals to represent attributes and relations approximately. For an attribute a in a relation, we represent its approximation $\bar{a}$ using a pair of attributes with the same domain, $(l_{\bar{a}}, u_{\bar{a}})$ with $l_{\bar{a}} \leq u_{\bar{a}}$. The values of these attributes are respectively the lower and upper bound estimates for the corresponding values of a. In our example in Figure 2, the attribute resulting from the `MAX` aggregation on `MalePopulation` was represented using intervals such as (10 billion, 20 billion), (12 billion, 13 billion), (12.8 billion, 12.9 billion) and so on.

A tuple is approximated using approximations for the respective attributes. Similarly, a relation R is approximated using a pair of relations $(L_{\bar{R}}, U_{\bar{R}})$ with $L_{\bar{R}} \subseteq U_{\bar{R}}$. Intuitively, $L_{\bar{R}}$ is the set of tuples that are guaranteed to be in R and $U_{\bar{R}}$ is the set of tuples that are potentially in R. With slight abuse of notation, we say

$$L_{\bar{R}} \subseteq R \subseteq U_{\bar{R}} \tag{1}$$

to denote that the approximate relations bound the exact R from below and above. Of course, R, $L_{\bar{R}}$ and $U_{\bar{R}}$ all have the same schema, with the caveat that the latter use approximate versions of the attributes.

For instance, consider a query that selects all the cities in the `Census` relation (shown in Table 1) with `MalePopulation`

Object	Exact	Approximate	Remarks
Attribute	a	$\bar{a} = (l_{\bar{a}}, u_{\bar{a}})$	$l_{\bar{a}}$: lower bound, $u_{\bar{a}}$: upper bound
Tuple	t	$\bar{t} = \langle \bar{a} \vert a \in t \rangle$	Approximations of all the attributes $\bar{t}(\bar{a})$: value of attribute $\bar{a}$ in tuple $\bar{t}$
Relation	R	$\bar{R} = (L_{\bar{R}}, U_{\bar{R}})$	$L_{\bar{R}} \subseteq R \subseteq U_{\bar{R}}$
Projection operator	$\pi_c(R)$	$\bar{\pi}_c(\bar{R}) = (\pi_c(L_{\bar{R}}), \pi_c(U_{\bar{R}}))$	Semantics unchanged
Rename operator	$\rho_{c_o \to c_n}(R)$	$\bar{\rho}_{c_o \to c_n}(\bar{R}) = (\rho_{c_o \to c_n}(L_{\bar{R}}), \rho_{c_o \to c_n}(U_{\bar{R}}))$	Semantics unchanged
Comparison operators	$=$ $\leq$	$(=_l, =_u)$ $(\leq_l, \leq_u)$	$\cdot_l$ *every* pair of exact values satisfies comparison $\cdot_u$ *some* pair of exact values satisfies comparison
Predicate	p	$\bar{p} = (\bar{p}_l, \bar{p}_u)$	Using $\cdot_l$ and $\cdot_u$ for all operators, $\bar{a}$ for all attributes a
Selection operator	$\sigma_p(R)$	$\bar{\sigma}_{\bar{p}}(\bar{R}) = (L, U)$	$L \subseteq \sigma_{\bar{p}_l}(L_{\bar{R}}) \subseteq \sigma_{\bar{p}_u}(U_{\bar{R}}) \subseteq U$
Cross-product operator	$R \times S$	$\bar{R} \bar{\times} \bar{S} = (L, U)$	$L \subseteq L_{\bar{R}} \times L_{\bar{S}} \subseteq U_{\bar{R}} \times U_{\bar{S}} \subseteq U$
Aggregation function	$f : \mathcal{B}(D_I) \to D_O$	$f_l, f_u : \mathcal{B}(D_I \times D_I) \to D_O$	$f_l(\mathbf{b}) = f(\{\{\ l \mid (l,u) \in \mathbf{b}\ \}\})$ $f_u(\mathbf{b}) = f(\{\{\ u \mid (l,u) \in \mathbf{b}\ \}\})$
Group By operator	${}_c\gamma_{g=f(a)}$	${}_c\bar{\gamma}_{\bar{g}:=f(\bar{a})}$ $\bar{g} = (l_{\bar{g}}, u_{\bar{g}})$	f: monotonically increasing with input weight $l_{\bar{g}} \leq \min_{\mathbf{b}} f_l(\mathbf{b}) \leq \max_{\mathbf{b}} f_u(\mathbf{b}) \leq u_{\bar{g}}$ (see Section 3.5)
DAQ scheme		Σ	Collection of all of the above
Relational algebra expression	$R := E$	$\bar{R} := \bar{E}_\Sigma$	Replace operators, operands with approx. from Σ
Online DAQ scheme		$\Sigma^{(m)}$	Indexed sequence of DAQ schemes (see Section 3.6)

Table 2: Summary of the notation used in Section 3

less than 1 million. Let S denote the output of this query. Then, the exact value of S is {`Tatooine, Rivendell`}, and one possible approximate estimate of S is $L_{\bar{S}} =$ {`Rivendell`} and $U_{\bar{S}} =$ {`Tatooine, Rivendell, Shire`}.

For an exact relation, we can define the equivalent approximate version by setting the same value for lower and upper bounds of the intervals (for every attribute and relation). We indicate this as $L_{\bar{R}} = U_{\bar{R}} \equiv R$.

3.2 Projection and Rename Operators

These projection and rename operators remain semantically unchanged.

$$\bar{\pi}_c(\bar{R}) := (\pi_c(L_{\bar{R}}), \pi_c(U_{\bar{R}}))$$
$$\bar{\rho}_{c_o \to c_n}(\bar{R}) := (\rho_{c_o \to c_n}(L_{\bar{R}}), \rho_{c_o \to c_n}(U_{\bar{R}})$$

As usual, c, c_o and c_n are lists of attributes, with c and c_o drawn from the schema of R and c_n a list of new attribute names ($|c_o| = |c_n|$).

3.3 Selection Operator

To define the selection operator, we first define how Boolean predicates are evaluated approximately. Since the tuples consist of approximate attributes represented as intervals, we use a 3-valued logic appropriate for interval comparisons. Our approach is to define a new pair of Boolean operators that correspond to tuples *certainly* and *potentially* satisfying the comparison. If *all* pairs of values drawn from the two intervals in the tuple satisfy the comparison, the tuple *certainly* satisfies it. If *some* pair of values drawn from the two intervals in the tuple satisfies the comparison, then the tuple *potentially* satisfies it.

For instance, corresponding to the equality comparison operator $=$, we define two operators $=_l$ and $=_u$, which conceptually behave like lower and upper bounds for the equality operator. A tuple $\bar{t}$ with attributes $\bar{a}$ and $\bar{b}$ satisfies $\bar{a} =_l \bar{b}$ if $\bar{t}(\bar{a}) = \bar{t}(\bar{b})$, i.e. if the lower and upper bounds for the interval values for attributes $\bar{a}$ and $\bar{b}$ in tuple $\bar{t}$ are all equal to each other. A tuple $\bar{t}$ satisfies $\bar{a} =_u \bar{b}$ if the intervals overlap at least partially, i.e. if $u_{\bar{a}} \in [l_{\bar{b}}, u_{\bar{b}}]$ or $l_{\bar{a}} \in [l_{\bar{b}}, u_{\bar{b}}]$ in $\bar{t}$. Note that $\bar{t}$ satisfies $=_l$ only when *every* possible exact tuple in the approximation interval satisfies $=$, and $\bar{t}$ satisfies $=_u$ when *some* possible exact tuple in the approximation interval could satisfy $=$.

Similarly, we extend $\leq$ operator as follows: $\bar{t}$ satisfies $\bar{a} \leq_l \bar{b}$ if $u_{\bar{a}} \leq l_{\bar{b}}$ in $\bar{t}$ (i.e. the entire interval for $\bar{a}$ lies below the interval for $\bar{b}$) and satisfies $\bar{a} \leq_u \bar{b}$ if $l_{\bar{a}} \leq u_{\bar{b}}$ in $\bar{t}$ (i.e. if the intervals for $\bar{a}$ and $\bar{b}$ overlap).

To illustrate these operators, consider the following query:

```
SELECT Country, AVG(MalePopulation) as M,
       AVG(FemalePopulation) as F
FROM Census GROUP BY Country
```

Table 3 shows an approximate estimate $\bar{C}$ of the result relation C for the above query. $U_{\bar{C}}$ consists of all the tuples shown, whereas $L_{\bar{C}}$ consists of only those that are marked with an *. (A practical DAQ implementation can use this technique to maintain only one table to represent both the bounds.) The tuple for `Narnia` has the approximate estimates for the attributes $\bar{M} = (100, 250)$ and $\bar{F} = (200, 300)$, indicating that the exact values M and F lie in these intervals. To check whether $M = F$ is satisfied for `Narnia`, applying the definitions above, we find that $\bar{M} =_l \bar{F}$ is not true (since we cannot guarantee that M and F are equal based only on this approximation) and $\bar{M} =_u \bar{F}$ is true (since it is potentially possible for M and F to be equal, as the intervals overlap). Similarly, $\bar{M} \leq_l \bar{F}$ is not true (since we cannot guaranteed that M and F will satisfy $M \leq F$) while $\bar{M} \leq_u \bar{F}$ is true (since there are some possible values for M and F that satisfy $M \leq F$). It is easy to verify that for `Discworld`, both $\bar{M} =_l \bar{F}$ and $\bar{M} =_u \bar{F}$ are true, and for `Westeros`, both $\bar{M} \leq_l \bar{F}$ and $\bar{M} \leq_u \bar{F}$ are true.

We define all other comparison operators similarly. Constant operands are approximated using a trivial interval (both bounds equal to the constant). The Boolean `OR` ($\vee$) and `AND` ($\wedge$) operators are applied to the Boolean *certain* and *potential* operators $\cdot_l$ and $\cdot_u$. This approach allows us to rewrite any first order predicate p in its approximate form $(\bar{p}_l, \bar{p}_u)$ by replacing all the attributes and constants by their approximate (interval) forms and replacing the comparison operators with their $\cdot_l$ and $\cdot_u$ forms respectively.

In our example, if p refers to the predicate $M \leq F$, then $\bar{p}$ is represented by the pair of predicates $(\bar{M} \leq_l \bar{F}, \bar{M} \leq_u \bar{F})$,

and the tuple for `Narnia` satisfies $\bar{p}_u$ but not $\bar{p}_l$.

The approximate form of selection operator σ with predicate p, $\bar{\sigma}_{\bar{p}}(\bar{R})$ is defined as a pair of relations (L, U) satisfying

$$L \subseteq \sigma_{\bar{p}_l}(L_{\bar{R}}) \subseteq \sigma_{\bar{p}_u}(U_{\bar{R}}) \subseteq U \tag{2}$$

Semantically, $\sigma_{\bar{p}_l}(L_{\bar{R}})$ is the relation consisting of exactly the tuples that *certainly* satisfy the predicate p on the intervals in $L_{\bar{R}}$, and $\sigma_{\bar{p}_u}(U_{\bar{R}})$ is a relation consisting exactly of all tuples that could *potentially* satisfy the predicate on the intervals in $U_{\bar{R}}$. These correspond to *exact* interval operations. L and U are themselves some *approximations* to these relations, to be defined by the exact scheme used for performing the operation. For instance, rounding the interval bounds up or down appropriately before performing the comparison would give us broader input intervals for the $\cdot_l$ and $\cdot_u$ operators, making L and U different from $\sigma_{\bar{p}_l}(L_{\bar{R}})$ and $\sigma_{\bar{p}_u}(U_{\bar{R}})$ respectively.

Let us again use $M \leq F$ as our predicate p, and $L_{\bar{C}}$ and $U_{\bar{C}}$ as defined in Table 3. It can be verified that $\bar{p}_l$ is true for `Westeros`, `Discworld` and `Pandora`, among which only `Discworld` and `Pandora` are in $L_{\bar{R}}$. Therefore, $\sigma_{\bar{p}_l}(L_{\bar{R}})$ consists of the tuples for `Discworld` and `Pandora`. Similarly, $\bar{p}_u$ is true for all of the tuples except `Essos`, and $\sigma_{\bar{p}_u}(U_{\bar{R}})$ therefore consists of all four tuples other than `Essos`.

Thus far in this example, we have evaluated the predicate and selection *exactly* on the approximate (interval) input. In general, however, a DAQ scheme may apply a further approximation at this stage if it achieves a better tradeoff between accuracy and responsiveness. For instance, it may be cheaper to evaluate the predicate after loosening the input interval by using just the single most significant digit of the attributes $\bar{M}$ and $\bar{F}$ (i.e., the hundreds' place). In this case, the result of $\bar{p}_u$ on $U_{\bar{R}}$ will be unaffected for all the tuples other than `Essos`. For `Essos`, the new widened intervals for both $\bar{M}$ and $\bar{F}$ will be (0,100), and $\bar{p}_u$ is now true for this tuple. The added flexibility from allowing L and U in our definition of $\sigma_{\bar{p}_l}(L_{\bar{R}})$ to be different from $\sigma_{\bar{p}_l}(L_{\bar{R}})$ and $\sigma_{\bar{p}_u}(U_{\bar{R}})$ enables a much wider selection of query plans (potentially choosing a different granularity for the intervals for each operator in the plan), in turn allowing optimal tradeoff between accuracy and responsiveness.

3.4 Cross-Product and Join Operators

We define the approximate cross-product operator between two relations $\bar{R}\bar{\times}\bar{S}$ as some pair of relations (L, U) such that

$$L \subseteq L_{\bar{R}} \times L_{\bar{S}} \subseteq U_{\bar{R}} \times U_{\bar{S}} \subseteq U \tag{3}$$

As in the discussion of Equation 2, L and U are approximations to $L_{\bar{R}} \times L_{\bar{S}}$ and $U_{\bar{R}} \times U_{\bar{S}}$, which are the results of the *exact* cross-product operations defined *approximate* relations R and S. The join operation is defined simply as a cross-product followed by a selection operation.

Country	**M**	**F**	*
Narnia	(100, 250)	(200, 300)	*
Westeros	(57, 76)	(82, 93)	
Essos	(808, 952)	(768, 784)	
Discworld	(431, 431)	(431, 431)	*
Pandora	(622, 843)	(845, 939)	*

Table 3: Approximation to the result relation C. Note that the input data, i.e. the instance of the `Census` relation is different from that shown in Table 1. Both `M` and `F` are approximate attributes, represented using intervals. All the tuples are in $U_{\bar{C}}$ and * indicates tuples that are also in $L_{\bar{C}}$.

3.5 Group By Operator

We now incorporate the group by operator γ into our framework. Note that the representation of intermediate results using just the lower and upper bounds prevents us from approximating arbitrary aggregation functions. Fortunately, as we show below, we can exploit some useful properties common to most of our usual aggregation functions to extend them to work with the interval representation.

In general, an aggregation function is of the form

$$f : \mathcal{B}(D_I) \rightarrow D_O \tag{4}$$

where D_I and D_O are the input and output domains, and $\mathcal{B}(D)$ is the set of all bags of elements drawn from D. We often have $D_I = D_O$, but not always. For instance, the aggregation functions `MAX`, `SUM`, `AVG` map numeric domains (integers or floating point numbers) into themselves, so that $D_I = D_O$. On the other hand, `TOP 100` maps numeric domains into 100-element vectors with elements from the same domain, and so D_I and D_O are not the same. Note that the aggregation functions are defined on bags instead of sets or lists, since the number of occurrences of a value in a column affect the result of `AVG`, `TOP k` etc, but these results are not dependent on the order of the input.

We assume that our domains have a partial ordering $\leq$, as is the case for common domains such as string, time and numeric. For the vector domain in the image (D_O) of the `TOP k` aggregation function, we use the l^2 norm of the vectors to define a partial ordering.

For a pair of bags $\mathbf{b_1}, \mathbf{b_2} \in \mathcal{B}(D)$, we define the comparison operation $\mathbf{b_1} \leq_w \mathbf{b_2}$ to be satisfied if $|\mathbf{b_1}| = |\mathbf{b_2}|$ and there is some bijection h from $\mathbf{b_1}$ to $\mathbf{b_2}$ satisfying $x \leq h(x)$ for all $x \in \mathbf{b_1}$. Intuitively, we can consider $\mathbf{b_2}$ to have been built element-by-element from $\mathbf{b_1}$ by either including the element unchanged or after increasing its value. When $\mathbf{b_1} \leq_w \mathbf{b_2}$, we say that $\mathbf{b_2}$ has a larger weight than $\mathbf{b_1}$. This gives us a partial ordering between bags from the same domain. Note that in an approximate attribute such as $\bar{M}$ in Table 3, the lower and upper bounds can be considered as two bags, the latter having a larger weight than the former.

An aggregation function f is said to be *monotonically increasing* with input weight if for all bags $\mathbf{b_1} \leq_w \mathbf{b_2}$, we have $f(\mathbf{b_1}) \leq f(\mathbf{b_2})$. It can be verified that the aggregation functions `AVG`, `SUM`, `MIN`, `MAX` and `TOP k` all satisfy this property, whereas `MEDIAN` and `STANDARD DEVIATION` do not.

Aggregation functions, as defined above, do not apply directly to the intervals representing approximate attributes. To extend an aggregation function of the form (4) to intervals, we define a pair of aggregation functions

$$f_l, f_u : \mathcal{B}(D_I \times D_I) \rightarrow D_O$$
$$f_l(\mathbf{b}) = f(\{\{\ l \mid (l,u) \in \mathbf{b}\ \}\})$$
$$f_u(\mathbf{b}) = f(\{\{\ u \mid (l,u) \in \mathbf{b}\ \}\})$$

where $\{\{\cdot\}\}$ denotes bags. f_l and f_u compute the aggregation f on the lower and upper bounds of the intervals in the bag $\mathbf{b}$. Note that since

$$\{\{\ l \mid (l,u) \in \mathbf{b}\ \}\} \leq_w \{\{\ u \mid (l,u) \in \mathbf{b}\ \}\}$$

we can conclude that $f_l(\mathbf{b}) \leq f_u(\mathbf{b})$ when f is monotonic with increasing input weight.

The approximate group by operator ${}_c\bar{\gamma}_{\bar{g}:=f(\bar{a})}$ takes as input a list of group by columns c from the schema of R as well as a monotonic aggregation function f on some attribute $\bar{a}$, and returns a pair of relations (L, U) as before, satisfying certain criteria. Naturally, L and U must have schemas $c \bigcup \{\bar{g}\}$ and must satisfy the functional dependency $c \rightarrow \{\bar{g}\}$. The values in the group by attributes must also satisfy the containment relation

$$\pi_c(L) \subseteq \pi_c(L_{\bar{R}}) \subseteq \pi_c(U_{\bar{R}}) \subseteq \pi_c(R)$$

i.e. they must be drawn from the respective lower and upper bound relations $L_{\bar{R}}$ and $U_{\bar{R}}$. The interval $(l_{\bar{g}}, u_{\bar{g}})$ of the aggregate attribute $\bar{g}$ in every tuple of L and U must respectively be lower and upper bounds for the aggregation f applied to all possible bags of a values in $L_{\bar{R}}$ and $U_{\bar{R}}$. We now use the monotonicity of f with increasing input weight to derive these bounds. For each set of values $\mathbf{c}$ taken by the group by attributes c in $L_{\bar{R}}$ and $U_{\bar{R}}$, let us denote the corresponding bags of interval values of $\bar{a}$ by $\mathbf{b}_{\mathbf{c},\bar{a}}$ and $\mathbf{B}_{\mathbf{c},\bar{a}}$ respectively. Note that $\mathbf{b}_{\mathbf{c},\bar{a}} \subseteq \mathbf{B}_{\mathbf{c},\bar{a}}$ as $L_{\bar{R}} \subseteq U_{\bar{R}}$ by definition. The bounds we seek are the minimum and maximum values taken by f on any bag $\mathbf{b}$ "between" these, i.e. $\mathbf{b}_{\mathbf{c},\bar{a}} \subseteq \mathbf{b} \subseteq \mathbf{B}_{\mathbf{c},\bar{a}}$. We know that $f_l(\mathbf{b}) \leq f_u(\mathbf{b})$ for any bag $\mathbf{b}$ of intervals. It follows that

$$\min_{\mathbf{b}} f_l(\mathbf{b}) \leq \max_{\mathbf{b}} f_u(\mathbf{b})$$

for any set of bags $\mathbf{b}$. So we require that

$$l_{\bar{g}} \leq \min_{\mathbf{b}} f_l(\mathbf{b}) \leq \max_{\mathbf{b}} f_u(\mathbf{b}) \leq u_{\bar{g}}$$

where $\mathbf{b}$ ranges over all bags $\mathbf{b}_{\mathbf{c},\bar{a}} \subseteq \mathbf{b} \subseteq \mathbf{B}_{\mathbf{c},\bar{a}}$. This ensures that $l_{\bar{g}}$ and $u_{\bar{g}}$ are deterministic lower and upper bounds for the aggregate g.

3.6 DAQ Scheme

A DAQ scheme Σ is specified by defining the relational operators $\bar{\pi}, \bar{\rho}, \bar{\sigma}, \bar{\times}$ and $\bar{\gamma}$, along with the comparison operators $=_l, =_u, \leq_l, \leq_u$, and aggregation functions etc. This provides us a closed algebra, where every relational operator operates on and returns approximate relations with approximate attributes, all using lower and upper bounds forming deterministic intervals. Given an expression E in standard relational algebra, we can convert into the corresponding DAQ expression in Σ, $\bar{\mathrm{E}}_\Sigma$, by replacing the operators and operands by their approximate forms.

We now define the notion of an *online* DAQ scheme. Intuitively, we want an online scheme to iterate on a database of relations, and produce intermediate results such that successive iterations bring us closer to the exact result.

More formally, we define an online scheme Σ as an indexed sequence of DAQ schemes $\Sigma^{(m)}$. When a relation is defined by a relational algebra expression $R :=$ E , we iteratively evaluate $\bar{R}^{(m)} := \bar{\mathrm{E}}_{\Sigma^{(m)}}$ for $m \in \mathbb{Z}^+$ until we converge on the exact result. These successive approximations $\bar{R}^{(m)}$ must satisfy the following conditions.

- *Monotonicity*: The bounds in the approximations must get progressively tighter.

$$L_{\bar{R}^{(m)}} \subseteq L_{\bar{R}^{(m+1)}} \subseteq U_{\bar{R}^{(m+1)}} \subseteq U_{\bar{R}^{(m)}}$$

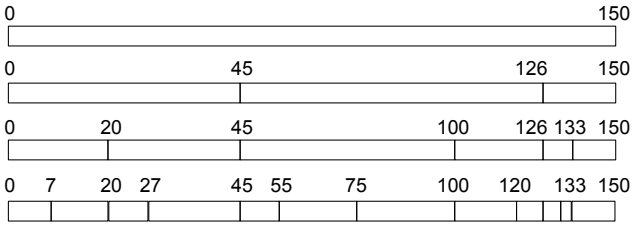


Figure 3: Conceptual DAQ attribute representation

The same must be true for all attributes $\bar{a}$ in all the tuples.

$$l_{\bar{a}}^{(m)} \leq l_{\bar{a}}^{(m+1)} \leq u_{\bar{a}}^{(m+1)} \leq u_{\bar{a}}^{(m)}$$

- *Convergence*: The iterations must eventually converge, i.e. for every input expression E, there is some $M \in \mathbb{Z}^+$ such that

$$L_{\bar{R}^{(M)}} = U_{\bar{R}^{(M)}}$$

and all attributes $\bar{a}$ in all tuples satisfy

$$l_{\bar{a}}^{(M)} = u_{\bar{a}}^{(M)}$$

and, this convergence must be to the exact relation R, i.e.

$$L_{\bar{R}^{(M)}} = U_{\bar{R}^{(M)}} \equiv R$$

The example we showed in Figure 2 is an online aggregation scheme where successive iterations examine more digits of the column and tighten the approximation interval for the attribute. The error in the approximation falls monotonically (in fact, exponentially) and both the bounds converge to the exact value after all the digits have been examined.

4 Designing DAQ Schemes

We now highlight some principles for designing efficient DAQ schemes. In this work, we limit our focus to single-relation queries without group-by clauses (i.e. queries with selection, projection and aggregations without group-by columns).

To efficiently evaluate range predicates as well as aggregates such as `MAX`, `MIN` and `TOP k`, the data structure we use to represent attributes must respect the order of values in the domain. Further, for online evaluation with iteratively improving approximation bounds, it is natural to seek a hierarchical data structure where different levels of the hierarchy are at different granularities and yield increasingly precise representations of the exact values. Thus, the key to efficient DAQ schemes is a hierarchical decomposition of the domain of the attribute into non-overlapping intervals, with successive levels of the hierarchy defining finer sub-partitions. The end-points of the intervals at a given level act as lower and upper bounds for all the values falling within it (see Figure 3). We also require an efficient way to find all the tuples in which the attribute takes values in a given interval (partition) at some level of the hierarchy.

One such data structure is Count B-tree (described in [13]) shown in Figure 4, a B-tree where each node also stores the minimum and maximum values, as well as count of leaves of the sub-tree rooted at the node. Computing `MAX` and `MIN` involves a traversal of a single path down the tree. `SUM` and `AVG` can be approximated using the bounds and counts for the nodes at successive levels of the tree.

In the next section, we explore an alternative scheme, viewing a bitsliced index representation of a column as a repeated binary partitioning of the underlying domain.

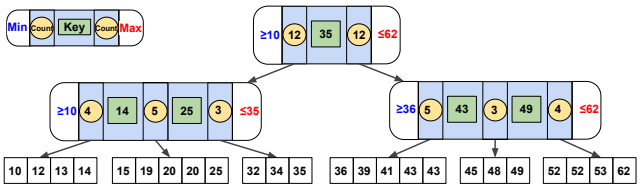


Figure 4: Count B-tree

5 The Bitwise DAQ Scheme

The Bitwise DAQ scheme is conceptually similar to the aggregation in Figure 2: in each iteration, we examine and compute the next most significant *bit* of the attribute.

Recall that the query that we used there computed the maximum value of the `MalePopulation` attribute. In Table 1, this maximum value was 12,876,461,144. Let us assume that the unsigned integer data type used for this attribute uses an n-bit representation (with $n = 32$).

Suppose that the user specifies a tolerance level for approximation error of 100,000, i.e. an estimate such as "between 12,876,400,000 and 12,876,500,000" is acceptable. We can use this fact to ignore the computation of the last 5 digits in the answer, i.e. the last 16 bits. In Section 5.3, we will show how the Bitsliced Index representation can be used to perform this aggregation with a significant speedup by ignoring half the bits of our 32-bit attribute.

Furthermore, we do not need the user to specify the above tolerance level a priori. Instead, if we perform the computation in the order of the most significant bit first, and continuously update the result to display the latest available digits; the user can choose either to terminate the computation early or to wait to get the exact answer.

In general, if we perform the computation of `MAX` as above using the most significant m bits and ignoring the last $b = n - m$ bits, the maximum possible error is $2^b - 1$, which is $\lceil \log_{10}(2^b - 1) \rceil$ digits (using the ceiling function $\lceil . \rceil$). Conceptually, the Bitwise DAQ scheme sets all the least significant b bits to zeroes and ones to obtain lower and upper bounds respectively for each attribute value. Thus, we can represent the interval using only the lower bound (using m bits) and the number of bits ignored (i.e. b), greatly simplifying the algorithms to be described in Section 5.3. We can similarly bound the error incurred in other operations caused by ignoring the least significant bits, so that our result approximates the most significant bits exactly. In successive iterations of the online scheme, we increase m and get an exponential reduction in the approximation error.

Note that as a DAQ scheme, the notion of "error" in the Bitwise scheme is different from that in SAQ approaches: it is more akin to a rounding error (due to the fact that we use reduced precision for operands) rather than a sampling error. Hence, in this approach, we can forego the probabilistic notion of confidence intervals in favor of deterministic approximation error bounds, thereby simplifying semantics and presentation for lay users.

5.1 Error Bounds

As in the case of `MAX` above, it is easy to bound the error in approximating many other aggregation functions by reducing the precision of the input to the most significant m bits, ignoring the least significant b bits. Note that these are worst-case deterministic error bounds, agnostic to both the distribution and order of data being aggregated. The error

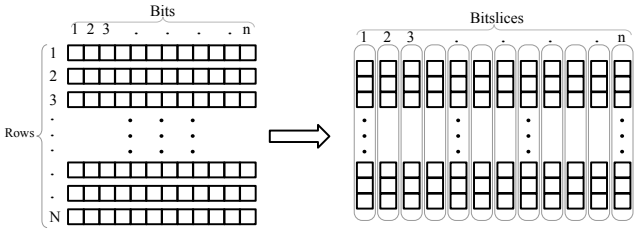


Figure 5: Bitsliced Index: N rows of an n-bit column are represented using n bitslices, each with N elements.

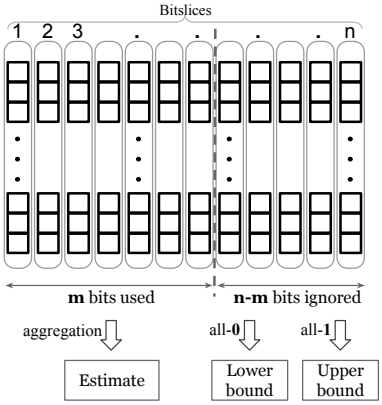


Figure 6: Bitsliced Index implementation

bound of $2^b - 1$ also applies to `MIN` and `AVG`. The approximation error when calculating the `SUM` for a column, however, scales with the number of tuples being aggregated (N).

$$\Delta_{\texttt{MAX}} = \Delta_{\texttt{MIN}} = \Delta_{\texttt{AVG}} = 2^b - 1$$
$$\Delta_{\texttt{SUM}} = N \times (2^b - 1)$$

Intuitively, this means that achieving the same error tolerance for `SUM` requires us to use more bits in the computation.

5.2 Pruning

A key advantage of aggregation using progressively increasing precision is that it permits early termination in the case of aggregation functions that are based on comparisons, allowing faster pruning of the search space. Both `MAX` and `TOP k` benefit greatly from this pruning effect, as the objective is to identify extreme values (outliers). Traditional sampling-based approaches do not approximate these well, particularly when the data is heavy-tailed. We will demonstrate this aspect further in Section 6.

5.3 Bitsliced Index Implementation

We call our specific implementation of the Bitwise approximation scheme discussed above as the *Bitsliced Index* implementation. It consists of the approximation algorithms discussed in the next section, executed on a bitsliced index representation of columns, i.e. an n-bit column is represented using an array of n bitslices (bitvectors), each with length equal to the number of rows in the table. The first bitslice contains the highest order bit of each row of the data column, second bitslice the next highest order bit, and so on (see Figure 5). This implementation uses the most significant m bitslices for aggregation and predicate evaluation ignoring the remaining $n - m$ bitslices. The lower and upper bounds are obtained using all zeroes and all ones for the least significant $n - m$ bits (see Figure 6).

Alternative DAQ scheme implementations using slices at byte boundaries, or using groups of smaller number of bits are possible, and rich targets for future work.

Converting a column into the bitsliced index representation (or vice versa) is an expensive operation. So there are advantages to maintaining both representations simultaneously for base data (i.e. not for intermediate results during query processing), and to process as much of the query without conversion as possible.

5.4 Bitslice Algorithms

We now provide algorithms for evaluating predicates and aggregations in the Bitsliced Index implementation of the Bitwise DAQ scheme. Note that we do not explicitly identify the intervals used to approximate the attribute values in the algorithms below: the estimate up to m bits, along with the number of bits ignored $(n - m)$, completely define the interval. Similarly, an approximate relation resulting from a predicate evaluation does not need to be explicitly identified using a pair of relations: since the input relation is a superset of both the lower and upper bound relations ($L_{\bar{R}}$ and $U_{\bar{R}}$ in our notation), we only need to maintain bitvectors corresponding to the membership of the tuples in the corresponding bounding relations.

Algorithm 1 shows how a predicate (here $X < Y$ for columns X and Y in bitsliced index representation) can be evaluated progressively. Using the two columns in the bitsliced index representation, it iteratively computes bitvectors that specify whether the tuples satisfy the predicate *potentially* or *certainly*, based on the most significant m bits. As discussed in Section 3.3, the predicate is *potentially* satisfied by a tuple if *some* pair of values from the two intervals corresponding to the approximate values of X and Y could satisfy it, and it is *certainly* satisfied if *every* pair of values from the two intervals satisfies it. The intervals here correspond to the values obtained by setting the least significant $n-m$ bits in the range from all zeroes to all ones. We introduce bitvectors `Equal` and `Less` to identify tuples for which $X = Y$ and $X < Y$ up to m bits. Note that given our choice of intervals, `Potential` is made up of tuples that satisfy either `Equal` or `Less`, and `Certain` is identical to `Less`.

We begin with all the tuples satisfying `Equal` and none satisfying `Less`, since without examining any of the bits, both intervals are identically equal to the entire domain. The two

Input : X, Y: Columns in bitsliced index representation
Output: Certain, Potential: Bitvectors of length NumElements representing the rows that *certainly* (respectively *potentially*) satisfy X < Y, when both columns have all their elements rounded to most significant m bits

```
SetZeroes(Less);
SetOnes(Equal);
for i ← 1 to m do
    Less  ← Less ∨ (Equal ∧ ¬X[i] ∧ Y[i]);
    Equal ←          Equal ∧ ¬(X[i] ⊕ Y[i]);
end
Potential ← Less ∨ Equal;
Certain   ← Less;
```

Algorithm 1: *Less Than* predicate up to m bits

Input : X: A column in bitsliced index representation
Output: Avg: Average of elements in X, rounded to m bits each

```
Sum ← 0;
for i ← 1 to m do
    Sum ← Sum ≪ 1 + CountOnes(X[i]);
end
Sum ← Sum ≪ (NumBitsPerElement − m);
Avg ← Sum ÷ NumElements;
```

Algorithm 2: *Average* aggregation up to m bits

bitvectors `Equal` and `Less` are then iteratively updated using successive bitslices from X and Y. When examining the i^{th} bit, a tuple satisfies $X < Y$ up to i bits if it already did so up to $i-1$ bits, or if it satisfied $X = Y$ up to $i-1$ bits and the i^{th} bits of X and Y in the tuple are 0 and 1 respectively. Similarly, a tuple satisfies $X = Y$ up to i bits if it already did so up to $i - 1$ bits and the i^{th} bits are equal (logical xnor). We omit the algorithms for other logical predicates, as they can be derived in a similar way.

The next three algorithms, Algorithms 2, 3 and 4, show how the bitsliced index representation for a column `X` can be used to evaluate three different aggregation functions on it: `AVG`, `MAX` and `TOP k` respectively. In Algorithm 2, we compute the column sum using the number of set bits in each of the first m bitslices, summed after weighting by the appropriate place values (power of 2). When divided by the number of rows, the resulting average is correct to m most significant bits, and the remaining $n - m$ bits can be set to all zeroes and all ones to obtain the lower and upper bounds of the approximation interval.

In Algorithm 3, we use a bitvector `PotentialMax` to represent all the maximal tuples after examining the most significant i bitslices of the column, i.e. the ones that could potentially have the maximum value, as they are not dominated by any others. Of course, all the tuples in this bitvector have identical values in this column, up to the most significant i bits. The algorithm begins by setting `PotentialMax` to all ones. Thereafter, after i iterations, we update `PotentialMax` to that subset of tuples that were already maximal after examining i bitslices and have a set bit in the $i + 1^{th}$ bitslice, as long as there is such a tuple. Note that we can terminate the algorithm before examining all m bitslices when we have only one maximal tuple remaining (see line **). If, however, there are multiple maximal tuples after m iterations, they are all equal to the actual maximum up to the first m bits and we make the arbitrary choice of picking the first one. This algorithm finds the index of the tuple having the approximate maximum value. Thereafter, there are many ways to recover the corresponding attribute value. In our prototype, we simply use the copy in the ordinary representation (a row or column store). The lower and upper bound of our estimate for the maximum follow, as before, by setting the remaining $n - m$ bits to all zeroes and all ones.

In Algorithm 4, we extend the above approach for estimating the maximum to finding Top k tuples. We use two bitvectors, `CertainTopK` and `PotentialTopK` as before, but in a break with our previous convention, `PotentialTopK` excludes the tuples in `CertainTopK`, as this makes the algorithms easier to describe. During the algorithm, `CertainTopK` identifies the `NumTopFound` $\leq$ `k` tuples that are maximal, and `PotentialTopK` identifies maximal tuples among the

```
Input  : X: A column in bitsliced index representation
Output: MaxIndex: Index of the tuple containing
        maximum of elements in X, rounded to m bits
        each
SetOnes(PotentialMax);
for i ← 1 to m do
    NextResult ← PotentialMax ∧ X[i];
    NumOnes ← CountOnes(NextResult);
    if NumOnes ≥ 1 then
        PotentialMax ← NextResult;
    end
**  if NumOnes = 1 then
        Break;
    end
end
MaxIndex ← IndexOfFirstSetBit(PotentialMax)
```

Algorithm 3: *Maximum* aggregation up to m bits

remaining, and we decrease the total count of tuples in the two bitvectors in successive iterations. In each iteration, we check how many of the tuples in `PotentialTopK` have a set bit in the next bitslice. If this count is less than the number we need (`k` − `NumTopFound`), then we add all of these tuples to `CertainTopK` and remove them from `PotentialTopK`. Otherwise, we simply remove all the others from `PotentialTopK`. We terminate the loop when the total count of tuples in both vectors falls to `k`, or when the m bitslices have all been examined. We then extract the indices and return them. If there are more than `k` tuples at this point, all the tuples in `PotentialTopK` are maximal up to m bits, in that the corresponding true Top k values (other than the ones in `CertainTopK`) would differ from these only in the least significant $n - m$ bits.

The break conditions labeled ** in Algorithms 3 and 4 allow us to terminate the loops as soon as we have found the required number of maximal elements, without iterating through the rest of the bitslices. Additionally, in our implementation, each bitslice is segmented into smaller portions of a few hundreds to thousands of bits, allowing us to prune more aggressively.

5.5 Extensions

In this section we describe two implementation extensions.

5.5.1 Parallelization

To parallelize the predicate evaluation and aggregation operators, we simply partition our column into a set of contiguous sub-columns and apply the algorithms from above in parallel to each sub-column. The results are then merged together according to the specifics of the operator. This technique is, of course, only applicable to non-holistic aggregates. For instance, to find the average `MalePopulation` across all cities in our `Census` example from Section 2, we can partition the `MalePopulation` column, find the partial sums for the sub-columns in parallel worker threads, add them together and divide by the total count. This approach has very little parallelization overhead, since there is virtually no synchronization required between worker threads. To further improve performance, we use a NUMA-aware assignment of partial aggregation work to worker threads, thus ensuring that each thread only reads its local data, and also to balance the load across cores in different sockets.

```
Input  : X: A column in bitsliced index representation
Input  : k: Number of top elements required
Output: TopKIndices: Indices of tuples containing the
        top k elements in X, rounded to m bits each
SetOnes(PotentialTopK);
SetZeroes(CertainTopK);
NumTopFound ← 0;
for i ← 1 to m do
    NextResult ← PotentialTopK ∧ X[i];
    NumOnes ← CountOnes(NextResult);
    if NumOnes ≤ k − NumTopFound then
        CertainTopK ← CertainTopK ∧ NextResult;
        PotentialTopK ← PotentialTopK ∧ ¬NextResult;
        NumTopFound ← NumTopFound + NumOnes;
    else
        PotentialTopK ← NextResult;
    end
**  if NumTopFound = k then
        Break;
    end
end
TopKIndices ← IndicesSet(CertainTopK);
PotentialTopKIndices ← IndicesSet(PotentialTopK);
PotentialTopKIndices ← First(PotentialTopKIndices, k
− NumTopFound);
Concat(TopKIndices,PotentialTopKIndices);
```

Algorithm 4: *Top* k aggregation up to m bits

5.5.2 Compression

Our methods can use compression to improve memory consumption and memory bandwidth utilization. For optimal performance, we need a compression scheme that can operate directly on the compressed codes, avoiding the decompression overhead. Intuitively, we expect that the most significant bitslices are likely to be highly compressible because column values on real datasets often follow skewed distribution or are in a limited range. But this is not always the case for the least significant bitslices. Our approach is to use Word-Aligned Hybrid compression scheme [16], as it dynamically adjusts to use a compressed (run-length encoded) representation or an uncompressed (literal) representation, for each sequence of bits in the bitslice longer than a single machine word. The resulting hybrid compressed bitslices can directly be used for bit operations such as logical `AND` and `OR` as well as `CountOnes`. In [13], we describe how dictionary compression can also be used with DAQ.

6 Evaluation

We evaluated our Bitsliced Index implementation of the Bitwise DAQ scheme both against the exact evaluation baseline in a columnar storage system, and a simple SAQ scheme.

The evaluation was performed using a synthetic dataset consisting of a single table `Census` of 10 million tuples with the columns `MalePopulation` and `FemalePopulation` represented using 32-bit unsigned integers. For different runs, these columns were drawn from four different distributions: Uniform random (over the entire 32-bit range, or over a small range of 10,000 numbers), and Zipf (with parameters 1.25 or 1.5, limiting to 32-bit values)[1]. The Zipf distribution was chosen as a representative heavy-tailed distribution,

[1]Some of the experimental results for data drawn from a smaller range are deferred to [13].

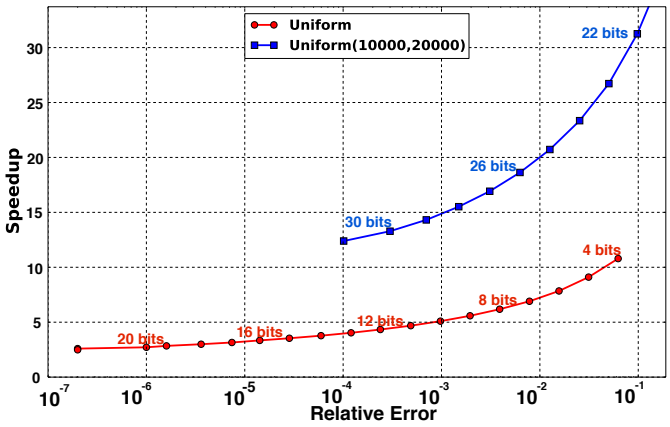


Figure 7: Speedup vs relative error tradeoff in evaluating a predicate on uniformly distributed data, varying no. of most significant bits used.

popularly used to model various data distributions, from populations of cities to sizes of organizations and word frequencies in natural language corpora. The parameters 1.25 and 1.5 mean that the most frequent 10 elements (the values 1 through 10) constitute respectively 52% and 77% of the observed data values, so most of the numbers (respectively 500,000 and 65,000 distinct values) are much larger than the mean. Unless otherwise mentioned, the results shown are from 5 runs each on 25 different randomly generated datasets, with each run taking between 5 and 20ms on a 2GHz Intel(R) Xeon(R) E5-2620 CPU. To keep this paper focused, we only consider in-memory settings as that environment is common for interactive analytics; i.e. all data sets are resident in main memory.

6.1 Predicate Evaluation

Figure 7 shows the tradeoff between speedup and relative error during evaluation of the following query:

```
SELECT COUNT(*) FROM Census
WHERE MalePopulation < FemalePopulation
```

We compared the implementation of Algorithm 1 to the baseline (exact) result of the predicate, with both columns uniformly distributed in the 32-bit range. The relative error on the x-axis is defined, as in Figure 2, by the ratio of the approximation error in the count to the exact count obtained in the baseline, i.e. the fraction of incorrectly evaluated predicates. The speedup on the y-axis is the ratio of execution cycles in the Bitsliced Index implementation to that in the baseline implementation. The curve is obtained by varying the number of most significant bits used in the evaluation from 4 to 32.

It can be seen that using fewer than 8 bits, the relative error already drops below 1%, while giving us a larger than factor of 6 speedup. As we increase the number of bits used in the evaluation, the error drops exponentially but the speedup also drops linearly. Beyond about 22 bits, the relative error drops to 0. This is a consequence of the pruning mentioned in Section 5.2. As we examine more bits, there are fewer tuples for which we are uncertain about whether the predicate is satisfied, and consequently there is less computation needed in the next iteration.

When the column values are drawn from a uniform distribution in a small range of 10,000 elements, the most significant bitslices are suppressed as all-zeroes or all-ones, and

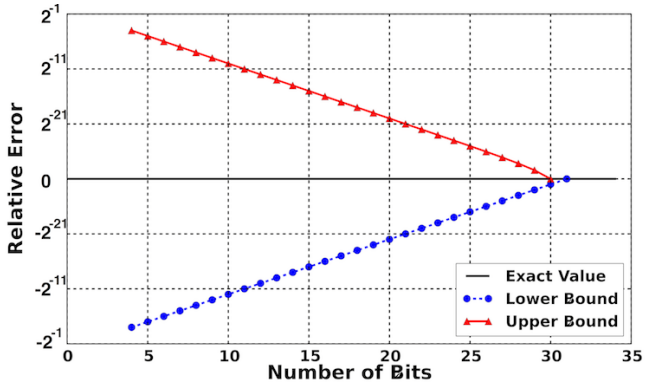


Figure 8: Relative errors in the lower and upper bounds during `AVG` for a uniformly distributed column.

the 1% relative error with a much larger speedup of a factor of 20.

With a more heavy-tailed distribution, such as the Zipf(1.25), nearly all the bits have to be examined before we can be relatively certain of the result, i.e. the pruning effect is minimal. In fact, for the error to drop below 1% at this dataset size, we have to examine all the bits. We therefore see a speedup of only 1.75x in this case.

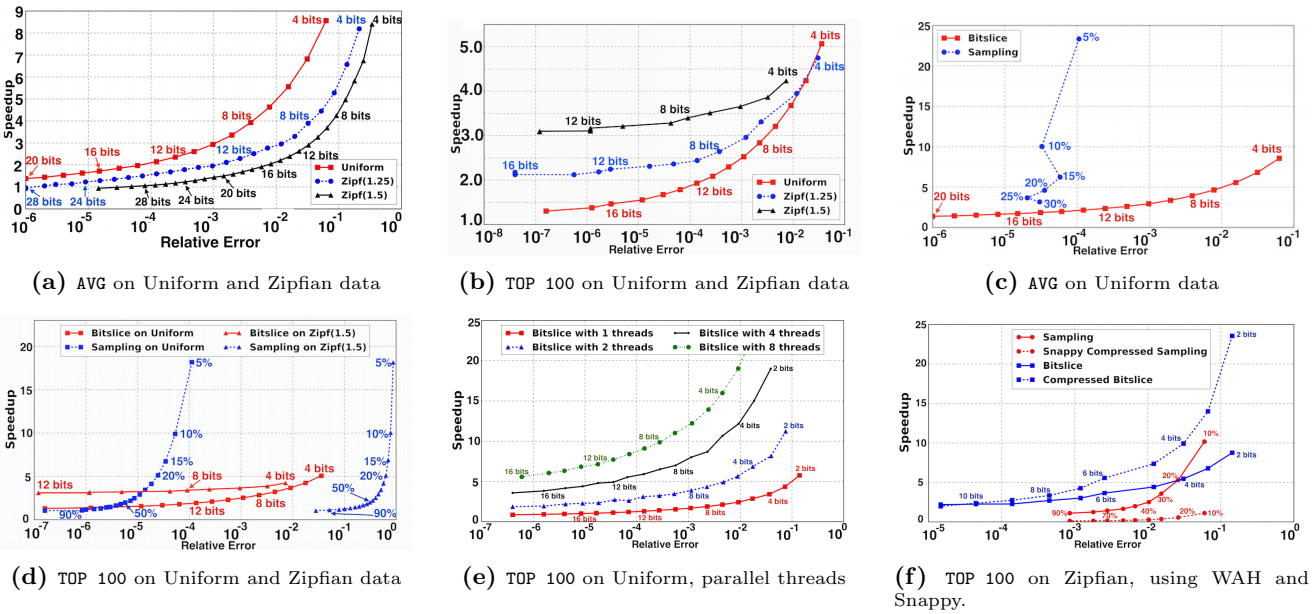
6.2 Aggregation: Compared to Baseline

We now consider the following aggregation query:

```
SELECT AVG(MalePopulation) FROM Census
```

For this experiment we use uniformly distributed data for the column `MalePopulation`. We use the same definition of relative error as before, i.e. the ratio of the error in our estimate of the average to its exact value obtained in the baseline. Figure 8 shows the relative error in the lower and upper bounds generated in the online evaluation of the Bitsliced Index scheme implementation of `AVG` on this column, as we vary the number of bits from 4 to 32 in a single sample run. The exact value is about 2^{31} (because of the uniform distribution in the 32-bit range). In this graph, 0 on the y-axis indicates the exact value. As the number of bits used increases, the error in our lower and upper bounds fall off exponentially. When using 10 bits, say, the lower and upper bounds are obtained by setting the remaining 22 bits to all zeroes or all ones, incurring errors of about 2^{21} in either direction. The relative errors are therefore $\frac{2^{21}}{2^{31}} = 2^{-10}$ in either direction.

Figure 9(a) shows the tradeoff between speedup and relative error (similar to Figure 7), using the `AVG` aggregation on each of the three distributions. The x-axis has the relative error for the lower bound, which is about the same as that for the upper bound (in absolute value). The speedup is measured against the baseline (exact) evaluation. As before, for the uniform distribution, fewer than 8 of the most significant bits are sufficient for a less than 1% error, yielding a speedup of a factor of 4 or more. Beyond about 20 bits, the error is negligibly small (not within range of figure). As the distribution becomes more heavy-tailed, while the time taken for aggregation (and hence the speedup) remains roughly the same for a fixed number of bits, the relative error in the estimate increases. In the extreme case, for the Zipf distribution with parameter 1.5, at least 16 bits need to be evaluated to obtain a 1% relative error, with a speedup of only a factor of 2.



(a) `AVG` on Uniform and Zipfian data

(b) `TOP 100` on Uniform and Zipfian data

(c) `AVG` on Uniform data

(d) `TOP 100` on Uniform and Zipfian data

(e) `TOP 100` on Uniform, parallel threads

(f) `TOP 100` on Zipfian, using WAH and Snappy.

Figure 9: Speedup vs relative error tradeoff, varying no. of most significant bits used in Bitslice and sample size in Sampling.

Next, we run the same experiment, but with the `TOP 100` query:

```
SELECT TOP 100 MalePopulation FROM Census
ORDER BY MalePopulation DESC
```

Now, the curves for the different distributions flip relative to each other. The result of the aggregation is a 100-element vector of the largest values in the `MalePopulation`, and the error on the x-axis here is the l^2-norm of the difference between the approximate and exact vector results, relative to the norm of the exact vector result. Here, even 6 bits suffice to give a relative error of less than 1% for the uniform distribution, with a speedup of about 3.5. A much higher speedup is obtained with the more heavy-tailed distributions, with the Zipf(1.5) distribution yielding greater than a factor of 4 speedup with 1% error using just 4 bits. While this aggregation function is clearly even more sensitive to extreme values, the pruning effect is largest for the heavy-tailed distributions. In fact, convergence to the exact result is reached after just 12 of the most significant bits are evaluated in Zipf(1.5).

We also ran the experiments using the `MAX` aggregation function. As expected, there is a clear gradation in the pruning effect and the impact of the heavy tail as we move from `AVG` to `MAX` to `TOP 100`.

6.3 Aggregation: Compared to SAQ

We now compare the Bitsliced Index implementation with a sampling-based approximation scheme. In these experiments, for the SAQ implementation, we computed the aggregation functions for sample sizes varying from 5% to 100% of the column `MalePopulation`, in increments of 5%. The Bitsliced Index implementation uses the entire dataset as before, without any sampling. The speedup numbers below are with respect to the baseline (exact) evaluation, and the relative errors are defined in the same way as before.

```
SELECT AVG(MalePopulation) FROM Census
```

When evaluating `AVG` for the uniform distribution (Figure 9(c)), the sampling approach clearly stands out. Even a 5% random sample gives a highly accurate (0.1% error) estimate of the average with a speedup of more than 20. Increasing the sample size has only a small impact on the error up to about 35%, after which the error drops to 0. On the other hand, as we described before, the Bitsliced Index aggregation only gives a factor of 5 speedup for a 1% error tolerance.

The relatively high performance of sampling-based approaches in this regime, where the aggregation function is insensitive to extreme values and the dataset is highly uniform, is expected. On the other hand, when we switch to a more outlier-sensitive aggregation function (`MAX` or `TOP 100`) and to a heavy-tailed distribution, Bitsliced Index aggregation starts to outperform sampling. This is shown in Figure 9(d), where the line style and color indicate the approximation scheme and the marker indicates the distribution of data for the `TOP 100` aggregation, using the following query:

```
SELECT TOP 100 MalePopulation FROM Census
ORDER BY MalePopulation DESC
```

As we move to the more heavy tailed (Zipf) distribution, the sampling implementation requires almost the entire dataset to achieve a low error rate, while the Bitsliced Index implementation performs much better.

6.4 Parallelization

Figure 9(e) shows how increasing the number of parallel worker threads allows us even greater freedom in choosing between improved accuracy of approximation and speedup relative to the (serial) baseline. For the case of `TOP 100` aggregation on 100 million uniformly distributed numbers, we see a nearly linear speedup upto 8 threads on our machine. Beyond this point, and for simpler aggregation functions like `SUM`, we see a significantly sub-linear scaling with the number of threads as we run into memory bandwidth limits.

6.5 Compression

Figure 9(f) shows that WAH compression improves the performance of the Bitsliced Index implementation, particularly for skewed distributions such as the Zipf(1.25) used in the figure. When the column values are drawn from a uniform

distribution over the entire 32-bit range, there is almost no compression possible, and the overhead of the WAH scheme results in worse performance than the Bitslice implementation without compression. As we increase the skew (and hence, compressibility) of the distribution to the Zipf distributions, we see a relative improvement in speedup of the compressed Bitslice implementation. In fact, for Zipf(2.0), we see a greater than 100x speedup over the baseline.

7 Related Work

Deterministic aggregation schemes: The notion of deterministic error guarantees has been proposed for some aggregation schemes in the past. For instance, Multi-Resolution Aggregation trees [8] is a multi-dimensional extension of the Count B-trees we presented in Section 4. However, unlike the Bitsliced Index scheme, they can not be used when the predicates are on columns that are not part of the index, and the extra cost of progressive convergence to the exact result can be prohibitive. Pirk et al. [11] use bitwise decomposition and rounding-based approximation, similar to the Bitsliced Index scheme, but in the context of optimal utilization of hardware heterogeneity.

To the best of our knowledge, our work is the first to highlight the importance of semantic (deterministic) guarantees and our theoretical framework conceptually encapsulates previously proposed schemes.

Bitsliced Index: Our Bitsliced Index scheme draws heavily from [10] and [14], but our work is differentiated by our focus on progressive approximation.

SAQ schemes: There is a rich history of SAQ schemes. Olken and Rotem [9] show how sampling can be used to answer interactive relational queries, and Hellerstein et al. [6] show how the confidence interval estimates can be derived in online aggregation. AQUA [1] and BlinkDb [3], among others, are notable prototypical examples of systems using SAQ scheme for interactive querying. An important problem in SAQ schemes is the derivation of confidence intervals (or, put differently, the required sample size for a given error tolerance level). Both the closed-form confidence interval estimates [5] as well as bootstrapping methods [12] suffer from some shortcomings [2], either in their applicability to aggregation functions that are sensitive to outliers, or in their reliability and efficiency when applicable. Recent analytical methods [17] show promise in overcoming the latter shortcoming.

Probabilistic databases: A lot of research has focused on explicitly modeling uncertainty in probabilistic databases [15]. DAQ schemes can be thought of as efficient ways of approximately computing the *support* of the probability distribution modeling the uncertainty in attributes or tuples. An interesting related system is ProbView [7] which uses intervals to represent uncertainty. However, these intervals are on the probability associated with attributes or tuples, unlike the intervals on domains in our work.

8 Conclusion

Our main contribution in this work has been to propose a new approach of using deterministic guarantees for approximation, which, while complementary to the traditional sampling-based approach, overcomes some of its limitations. We formally defined the DAQ paradigm and developed a closed algebra capable of approximating relational queries. We also enumerated some desiderata for DAQ schemes and outlined an approach to develop them. We delved into one such scheme - the Bitsliced Index scheme - and provided efficient algorithms for evaluating predicates and aggregations in a specific Bitsliced Index implementation of this conceptual scheme. The experimental evaluations confirm that this scheme performs efficient approximation, giving estimates with less than 1% error along with a speedup of 6x for predicate evaluation and 2-4x for aggregation functions. In comparison with SAQ schemes, the Bitsliced Index implementation performs best when the data distribution has a heavy tail (i.e., when there are outliers in the dataset), and for aggregation functions that are sensitive to outliers.

The particulars of our Bitsliced Index implementation leave much room for improvement: we currently only handle unsigned integers and support a subset of the capabilities in SQL. As part of future work, we plan to explore methods to extend the DAQ framework to group-by and join queries. We also plan to explore query optimization methods to combine SAQ and DAQ schemes and/or switch between these two schemes based on the workload and data characteristics.

9 Acknowledgements

This work was supported in part by the National Science Foundation under grants IIS-0963993 and IIS-1250886, and a gift donation from Google.

10 References

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

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

[3] 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 *ACM EuroSys*, 2013.

[4] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 2007.

[5] P. J. Haas. *Hoeffding Inequalities for Join Selectivity Estimation and Online Aggregation*. IBM, 1996.

[6] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *ACM SIGMOD Record*, 1997.

[7] L. V. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 1997.

[8] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *ACM SIGMOD Record*. ACM, 2001.

[9] F. Olken and D. Rotem. Simple random sampling from relational databases. *VLDB*, 1986.

[10] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *ACM SIGMOD Record*, 1997.

[11] H. Pirk, S. Manegold, and M. Kersten. Waste not? efficient co-processing of relational data. In *ICDE*. IEEE, 2014.

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

[13] N. Potti and J. M. Patel. DAQ: A New Paradigm for Approximate Query Processing (Extended Version) at `http://pages.cs.wisc.edu/~nav/DAQ-extended.pdf`. Technical report, University of Wisconsin-Madison, 2014.

[14] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001.

[15] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 2011.

[16] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 2006.

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