

# Towards Special-Purpose Indexes and Statistics for Uncertain Data\*

Anish Das Sarma, Parag Agrawal, Shubha U. Nabar, Jennifer Widom

{anish,paraga,sunabar,widom}@cs.stanford.edu  
Stanford University

**Abstract.** The Trio project at Stanford [35] for managing data, uncertainty, and lineage is developed on top of a conventional DBMS. Uncertain data with lineage is encoded in relational tables, and Trio queries are translated to SQL queries on the encoding. Such a layered approach reaps significant benefits in terms of architectural simplicity, and the ability to use an off-the-shelf query processing engine. In this paper, we present special-purpose indexes and statistics that complement the layered approach to further enhance its performance.

First, we identify a well-defined structure of Trio queries, relations, and their encoding that can be exploited by the underlying query optimizer to improve the performance using Trio’s layered approach. We propose several mechanisms for indexing Trio’s uncertain relations and study when these indexes are useful. We then present an interesting order, and an associated operator, which are especially useful to consider when composing query plans. The decision of which query plan to use for a Trio query is dictated by various statistical properties of the input data. We identify the statistical data that can guide the underlying optimizer, and design histograms that enable estimating the statistics accurately.

## 1 Introduction

The field of uncertain databases has received considerable attention for several decades now. While most prior work (e.g., [1, 5, 12, 16, 18, 24–27]) focuses on theoretical aspects, there has been recent interest in building systems [4, 8, 10, 29]. Motivated by a diverse set of applications such as data integration, deduplication, scientific data management, information extraction, and others, the Trio project at Stanford [29, 35] has been studying the combination of uncertainty and lineage as the basis for a new type of database management system.

In this paper, we study techniques for enhancing query optimization in Trio. The basic construct for uncertainty in Trio’s ULDB data model [6] is *alternatives*. Alternatives in a tuple specify a nonempty finite set of possible values for the tuple. For example:

$\boxed{(\text{Thomas, Main St.}) \parallel (\text{Tom, Maine St.})}$

---

\* This work was supported by the National Science Foundation under grants IIS-0324431 and IIS-0414762, by grants from the Boeing and Hewlett-Packard Corporations, by a Microsoft Graduate Fellowship, and by Stanford Graduate Fellowships from Cisco Systems and Sequoia Capital.

contains a tuple with two alternatives giving the two possible values for the tuple. The ULDB data model is defined formally in Section 2. The Trio system is layered on top of a conventional relational DBMS [29]. ULDB relations are encoded as conventional relational tables, and a rewriting-based approach is used for most data management and query processing.

The simple and elegant layered approach in Trio enables using an off-the-shelf query processing engine<sup>1</sup> (QPE) and its optimization capabilities. However, the performance of the layered approach can further improve if the QPE detects and exploits the inherent “structure” of encoded ULDB relations during query processing. This paper suggests novel techniques that augment the optimization capability of the layered approach’s QPE.

We show that the structure of ULDB relations merits building specialized index structures and associated access methods. While some of these index structures are equivalent to conventional indexes over encoded ULDB relations, some others are non-traditional indexes that cannot be created by a traditional QPE. We then show that query plans executing queries over Trio relations need to consider a special *interesting order*, namely that of grouping alternatives based on the tuple they are a part of. Consequently, we need an operator that performs the grouping. Such a grouping through the query plan eliminates the need for an expensive sort operation currently performed on the result in Trio.

Every Trio query can be answered using many query plans assembled from the new operators, indexes, and access methods mentioned above in conjunction with those already existing in the QPE. As in conventional query optimization, the choice of which query plan to use is based on an estimate of the cost of executing query plans, which critically depend on various forms of statistics and cardinalities maintained by the database. We enumerate several important statistics, and provide histograms that allow us to efficiently and accurately estimate the statistics. The techniques describe in this paper can be incorporated into the underlying QPE, while still maintaining the overall layered architecture adopted by Trio.

Finally, we present several interesting avenues for future work that our paper suggests. Although several systems have been built recently for managing uncertain data, none of them incorporate query optimization techniques similar to the ones described in this paper. Prior work in Trio itself has also focussed on other aspects such as modeling and design, confidence computation, and versioning. Obviously, query optimization in conventional databases is an extensively studied area, and we believe this first paper on query optimization for uncertain databases can form the basis for a lot of further work. Since many previously proposed data models for uncertainty (for example, [3, 5, 12, 17, 28, 32]) use similar constructs as ULDBs, the techniques described in this paper can be suitably adapted for these data models as well.

## 1.1 Contributions and Outline

The specific contributions of this paper are as follows:

---

<sup>1</sup> Our current implementation uses PostgreSQL

1. We provide techniques for indexing ULDB relations encoded as conventional relations, and describe the new access methods they yield. (Section 3)
2. We motivate the need for considering a new interesting order in query plans, and design an operator that ensures this order. (Section 4)
3. We enumerate various kinds of statistics necessary in choosing the optimal query plan from the set of all query plans combining conventional and new operators described above. We present histograms that enable estimating these statistics efficiently and accurately. (Section 5)
4. We discuss a slew of interesting and challenging problems our paper opens up, that we hope would form the basis for further research in the area. (Section 6)

Section 2 introduces our data model and its relational encoding, Section 7 presents related work, and we conclude with future work in Section 8.

## 2 Preliminaries

We first introduce Trio’s ULDB data model, then present the relational encoding of ULDBs in the layered approach, and finally describe the workload of queries we consider.

### 2.1 ULDB Data Model

We briefly introduce the ULDB data model here, and refer the reader to [6] for additional features and detailed semantics. Each tuple in a ULDB relation consists of a set of mutually-exclusive *alternatives*, each of which can be the tuple’s actual value. ULDB relations conform to the standard *possible-worlds semantics* [1, 6, 11, 31, 33]: A ULDB relation represents a set of possible worlds, each of which is an ordinary relation. The possible worlds for an uncertain relation are obtained by choosing one alternative value for each tuple, in all possible ways. For example, the following ULDB relation represents four possible worlds. (Alternatives are separated by ||.)

<b>R(Name, Address)</b>
(Thomas,Main St.)    (Thomas,Maine St.)
(Bill,Poplar Ave)    (William,Poplar Ave)

The ULDB data model also has other uncertainty constructs, such as “?” and confidence values, which are not crucial for optimization. It’s important to note that the Trio system decouples confidence computation from data computation [14], and hence confidence values are disregarded for the rest of the paper. Also, we don’t discuss the lineage feature in ULDBs (refer to [6]), as lineage is only a function of the query, and independent of the specific query plan used to compute the result.

### 2.2 Relational encoding

We now describe how the restricted ULDB data model described above is encoded in regular relational tables. We refer the reader to [7, 29] for complete details on encoding

ULDB databases. Hereafter we use *x-tuple* to refer to a tuple in the ULDB model, which includes alternatives, and we use *tuple* to denote a regular relational tuple.

Consider a ULDB relation  $T(A_1, \dots, A_n)$ . The data portion of  $T$  is stored as a conventional table referred to as  $T_{enc}$  with two additional attributes:  $T_{enc}(aid, xid, A_1, \dots, A_n)$ . Each alternative in the original ULDB table is stored as its own tuple in  $T_{enc}$ , and the additional attributes function as follows:

- **aid** is a unique alternative identifier.
- **xid** identifies the x-tuple that this alternative belongs to.

Relation  $R$  from the previous section would be encoded as follows.

xid	aid	Name	Address
1	11	Thomas	Main St.
1	12	Thomas	Maine St.
2	13	Bill	Poplar Ave
2	14	William	Poplar Ave

### 2.3 Queries

As mentioned before, queries over ULDB relations are translated to queries over the encoded relations. To obtain `xid`'s on the resulting relation, the alternatives of the result need to be grouped based on which x-tuple they are a part of. Therefore, all tuples in the result of the translated query are grouped by `xids` of the input relations. For example, if we perform a join of  $R$  and  $S$ , the translated query over  $R_{enc}$  and  $S_{enc}$  includes the clause “group-by  $R_{enc}.xid, S_{enc}.xid$ ”. Exact details of this translation are omitted, and can be found in [7, 29].

## 3 Indexing ULDB Relations

As described in Section 2, there is a special attribute `xid` in all Trio encoded relations, and all query translations involve a group by on `xid`. In this section, we introduce new indexes that are especially useful in the presence of this special attribute. We use a simple running example to ground our discussion.

Consider a relation  $R$ , and a selection query which does a range scan on the attribute  $A$ : “select \* from  $R$  where  $A \leq 5$ ”. The translated query groups the result by the `xid` attribute. The relation  $R_{enc}$  is often stored clustered on `xid`, but may also be clustered on other attributes, which may or may not be  $A$ . For the selection query on  $A$ , the following indexes may be useful:

- Index on  $A$   
 An index on  $A$  may be used to retrieve only the tuples in  $R_{enc}$  that satisfy the predicate  $A \leq 5$ . This index lets us efficiently retrieve all alternatives that satisfy the predicate, but now they need to be grouped to form x-tuples. A sort on `xid` is required to group the result alternatives into x-tuples. Such a query plan can be efficient for highly selective queries, i.e., the result contains very few alternatives, making the grouping step inexpensive.

- Index on  $xid$   
An index on  $xid$  lets us retrieve all alternatives in an  $x$ -tuple together; i.e., it returns tuples of  $R_{enc}$  in order of their  $xid$  values. This allows us to avoid the sort since result tuples are generated already grouped by  $xid$ . This index may be useful if the predicate is not very selective, especially if the data is stored clustered by  $xid$ .
- Index on  $(xid, A)$   
If  $x$ -tuples are very wide, i.e., contain a large number of alternatives, we may be able to use an index on  $(xid, A)$ , to only retrieve alternatives that satisfy the predicate. This also avoids the sort at the end, and thus may yield an efficient execution plan.
- Index on  $(A, xid)$   
For queries that use an equality predicate on  $A$ , it might be useful to use an index that returns all alternatives satisfying the predicate grouped by  $xid$ . This index allows us to avoid the sort which may be expensive for large results. But equality predicates seldom have large results. We would ideally like an index that also works for range queries, and still avoids the sort. Such an index is presented next.

The indexes discussed above help in either avoiding the sort required for the group-by on  $xid$ , or prune down the amount of data accessed by evaluating the predicate before retrieving the tuples. An index on  $(A, xid)$  accomplishes both objectives for equality predicates on  $A$ . We now describe a new index that generalizes this index for efficient range scans over relations stored clustered by  $xid$ .

- Index on  $A_x$   
An index on  $A_x$  refers to an index that retrieves all  $x$ -tuples that contain an alternative satisfying some predicate on  $A$ . We can then apply the predicate to each alternative of the  $x$ -tuple to keep only those that satisfy it. Although this index may often retrieve more tuple alternatives than an index on  $A$ , it can often be more efficient because it makes no more random accesses, and also avoids the need for a sort. The sort is avoided because the index guarantees an “interesting order” that has result alternatives grouped by  $xid$ .
- Index on  $(A_x, A)$   
For wide  $x$ -tuples, it may again be useful to have an index on  $A$  within an  $x$ -tuple. This index is similar to the index on  $(xid, A)$ , but prunes  $x$ -tuples and retrieves only those that contain at least one alternative that satisfies the predicate.

*Example 1.* We illustrate the benefit of  $A_x$  over the conventional indexes described before using an example. Again consider the query: “select \* from  $R$  where  $A \leq 5$ ”. Suppose  $R$  contains  $a$  alternatives satisfying the predicate, and suppose these  $a$  alternatives constitute  $x$   $x$ -tuples. Let us consider the cost of executing this query using four of the relevant indexes:

1. Index on  $A$ : We would use the index to get all the  $a$  alternatives using index lookups, and then these  $a$  alternatives would need to be grouped in memory based on their  $xids$ . Hence the total cost is:  $a$  index lookups + group  $a$  tuples.
2. Index on  $(A, xid)$ : Since the query involves a range scan, using the additional index on  $xid$  does not help us do the grouping along with the index lookups. Hence, the cost by using the index on  $(A, xid)$  is the same as the cost by using the index on  $A$ .

3. Index on `xid`: Using the index `xid` we scan the entire relation, and within each `x`-tuple we check whether there is any alternative with  $A \leq 5$ . The cost of this would be to do an index scan of the entire relation. The grouping step is saved because all alternatives are obtained grouped by `xids`.
4. Index on  $A_x$ : Finally consider using the index on  $A_x$ . We perform an index scan to obtain all `x`-tuples containing at least one alternative with  $A \leq 5$ . The cost of this step is an index scan of  $x$  `x`-tuples. We then need to filter all alternatives among these `x`-tuples that do not satisfy the predicate. Hence using the index  $A_x$ , we save the grouping step and still require looking only at `x`-tuples that have alternatives satisfying the predicate.

□

## 4 Query Plans

In this section, we discuss some new challenges that are encountered in creating query plans to answer queries over uncertain data. We start by discussing a new “interesting order” that can benefit many queries.

### 4.1 XGroup

Recall the index on  $A_x$  described above, which allows for an efficient access method to retrieve alternatives grouped by `xid`. For complicated queries, possibly including several joins, it may be useful to maintain this grouped order incrementally all through the query plan in order to avoid a massive `group-by` at the end. Intuitively, the idea is to have `x`-tuples flowing through operators instead of alternatives. By maintaining all alternatives of an `x`-tuple (nearly) together, we may get efficient executions of queries with large results. We now formally define the XGroup order. We shall see that XGroup is a fundamentally weaker requirement than sort on an attribute, and hence can often be maintained over the entire query plan without significant overhead.

**Definition 1 (XGroup).** *Let  $\mathcal{R} = \{R_1, \dots, R_n\}$  be the relations in the `from` clause of a query. The translated query contains a `group-by` on the `xid` attributes of each relations in  $\mathcal{R}$ . Suppose the subtree rooted at a node  $N$  in the query tree joins relations in  $\mathcal{R}_N \subseteq \mathcal{R}$ . If the tuples flowing out of  $N$  are guaranteed to be grouped by the `xids` of a set  $\mathcal{B}_N \subseteq \mathcal{R}_N$  of the relations, then  $N$  is said to guarantee the XGroup order on the set of relations  $\mathcal{B}_N$ . The final query result thus needs to be XGrouped on  $\mathcal{R}$ . □*

An index access on relation  $R$  using the index on  $A_x$  for some attribute  $A$  guarantees an XGroup on  $\{R\}$ . Other access methods that use an index on `xid` or  $(\text{xid}, A)$ , or that scan relations clustered by `xid` also guarantee an XGroup on  $\{R\}$ . Note that selection and projection operators also preserve XGroups; i.e., if the input to such an operator is XGrouped on  $\mathcal{B}$ , the output will also be XGrouped on  $\mathcal{B}$ .

Many join operators also preserve XGroups. For instance, in a nested join operator, suppose the inner set of input alternatives are XGrouped on  $\mathcal{B}_i$  and the outer are XGrouped on  $\mathcal{B}_o$ . It is easy to see that operators like nested-loop join (or nested-block

joins) and nested-loop index join preserve the XGroup on both the inner and the outer, i.e., the result alternatives will be XGrouped on  $\mathcal{B}_i \cup \mathcal{B}_o$ . Some other join operators may only preserve XGroups for one of the inputs. For instance a hash-join preserves the XGroup of the outer, i.e., returns alternatives XGrouped on  $\mathcal{B}_o$ . Many operators in traditional database query processors naturally preserve XGroup, making it efficient to maintain and utilize.

Recognizing and incrementally maintaining XGroups through a query plan can often help avoid an expensive `group-by` (usually implemented through a sort) at the end. As an extreme example, if all relations in a query are accessed using access methods that guarantee XGroups on input relations, and all operators preserve them, then no final `group-by` needs to be performed in the query plan. Our selection example earlier was the simplest case that illustrates how the `group-by` can be eliminated. Requiring all access methods and operators to preserve XGroups narrows the set of acceptable query plans to a very small set. Hence, we allow query plans to have combinations of access methods and operators that may or may not preserve XGroups. However, we can still benefit from partially XGrouped results in making the final `group-by` cheaper, through a new unary operator described next.

**Definition 2 (XGB).** *Let the input to operator XGB be alternatives XGrouped on  $\mathcal{B}_i$ . XGB returns the alternatives XGrouped on  $\mathcal{B}_o \supset \mathcal{B}_i$ .*  $\square$

The XGB operator is thus XGroup-aware ensuring that the result alternatives are XGrouped on a larger set of relations. It essentially breaks up each group in the input corresponding to one combination of `xids` of  $\mathcal{B}_i$  into even smaller groups, corresponding to combinations of `xids` of  $\mathcal{B}_o$ . The advantages of using this new operator are threefold: (1) In conjunction with an access method that provides the XGroup order, the XGB operator allows us to incrementally do the `group-by`. This may have lower cost than doing a single expensive `group-by` at the end of the query plan. (2) The operator only needs to look at one group in the input stream of alternatives at a time to construct the output grouping. If the groups in the input are not large, XGB can be performed very efficiently, operating in a non-blocking fashion. (3) For the same reason, the memory requirements for performing XGB is also considerably less than an operator that XGroups on  $\mathcal{B}_o$  assuming no input grouping.

*Example 2.* Consider a query that joins two relations  $R_1$  and  $R_2$  with  $n_1$  and  $n_2$  x-tuples respectively. Consider a query plan that joins alternatives from the two relations in any arbitrary order and then does a final `group-by` on  $R_1.xid, R_2.xid$ . The cost of this final `group-by` will be  $C_G(n_1n_2)$ , where  $C_G$  gives the expected cost of doing the `group-by` as a function of the total number of x-tuples. For instance, if the `group-by` is achieved by sorting the alternatives by  $R_1.xid, R_2.xid$  then  $C_G(n_1n_2)$  would be  $O(n_1n_2 \log(n_1n_2))$ . In contrast, if the output of the join is already XGrouped on  $R_1.xid$ , the cost of the final `group-by` is  $O(n_1C_G(n_2))$ , which is typically less than  $C_G(n_1n_2)$ . Intuitively, a problem of size  $n_1n_2$  is reduced  $n_1$  problems of size  $n_2$ .  $\square$

The presence of even one access method or operator that guarantees the XGroup order can thus reduce the cost of the `group-by`, if the optimizer recognizes the operators preserving XGroups.

## 4.2 Query Planning

In order to choose an optimal plan for executing a query, the query planner needs to decide which combination of the new indexes, operators introduced, and the traditional relational operators must be used for a given query. We motivate the need for estimating various statistics that would guide the query planner in choosing an efficient query plan.

Consider our example query from Section 3: “select \* from R where  $A \leq 5$ ”. We now look at each access method discussed in Section 3 emphasizing the relevant statistics that impact their execution cost.

- Index on A  
To estimate the cost of accessing the relation through an index on A, the optimizer needs to determine the number of alternatives that satisfy the predicate.
- Index on xid  
To estimate the number of x-tuples retrieved using an index on xid, the optimizer needs to know the total number of x-tuples in the relation R.
- Index on (xid, A)  
As discussed in Section 3, this index is useful only if x-tuples contain large number of alternatives. Hence, to decide whether or not to use this index, an estimate indicating the average number of alternatives per x-tuple (width of x-tuple) is useful. The average width of x-tuples satisfying the predicate might differ from the average width for the entire relation, and can yield more accurate cost estimates.
- Index on (A, xid)  
The number of alternatives that satisfy the predicate determines the cost of accessing the relation using an index on (A, xid). In addition, the number of x-tuples returned is determined by estimating the number of x-tuples that contain at least one alternative that satisfies the predicate on A.
- Index on  $A_x$   
The number of random accesses made in accessing the relation R using an index on  $A_x$  is determined by the number of x-tuples that contain at least one alternative satisfying the predicate.
- Index on ( $A_x$ , A)  
This index is useful in cases where the average width of an x-tuple is large, thus justifying indexing on A within an x-tuple. A useful statistic to determine the efficacy of the index is the average width of x-tuples that satisfy the predicate.

In the next section, we formally define the above statistics and provide methods for estimating them.

## 5 Statistics and Histograms

We will now formally define the different statistics from the previous section that guide the optimizer in choosing the optimal query plan. Some of these statistics can be exactly maintained, while others need to be estimated and we introduce new kinds of histograms for this purpose.

## 5.1 Exact Cardinalities

For each relation  $R$  in the database, the following global statistics can be exactly maintained:

- $X\text{-Card}(R)$ : The number of  $x$ -tuples in  $R$ . In terms of  $R$ 's encoding,  $X\text{-Card}(R)$  gives the number of distinct  $x$ ids in  $R_{enc}$ .
- $A\text{-Card}(R)$ : The number of alternatives in  $R$ .  $A\text{-Card}(R)$  translates to the number of tuples in  $R_{enc}$ .
- $AvgWidth(R)$ : The average number of alternatives per  $x$ -tuple in  $R$ , i.e.,  $\frac{A\text{-Card}(R)}{X\text{-Card}(R)}$ .

*Example 3.* Consider the following relation  $R$ .

<b>R(Name, Address, Salary)</b>
(Thomas, Main St., 50K)    (Thomas, Maine St., 50K)
(Bill, Poplar Ave, 35K)    (William, Poplar Ave, 40k)    (Billy, Poplar Ave, 40K)
(Alice, Euclid Ave, 10K)

Here  $X\text{-Card}(R) = 3$ ,  $A\text{-Card}(R) = 6$  and  $AvgWidth(R) = 2$ . □

Next we consider statistics that are infeasible to maintain exactly, and hence need to be estimated.

## 5.2 Alternative Counts

Consider a relation  $R$  with an attribute  $A$ . The following are statistics about alternatives that we would like to maintain for point and range queries:

- $A\text{-Selectivity}(R, A, v)$ : The number of alternatives in  $R$  that satisfy  $A = v$ .
- $A\text{-Selectivity}(R, A, x, y)$ : The number of alternatives in  $R$  that satisfy  $x \leq A \leq y$ .

*Example 4.* Consider the relation  $R$  from Example 3.  $A\text{-Selectivity}(R, \text{Name}, \text{Thomas}) = 2$ , whereas  $A\text{-Selectivity}(R, \text{Salary}, 40K, 60K) = 4$ . □

Clearly, the above selectivities translate to counting the number of tuples in  $R_{enc}$  satisfying  $A = v$  and  $x \leq A \leq y$  respectively. These cardinalities over the conventional relational table  $R_{enc}$  can be estimated using well-known sampling or histogram techniques. For example, in Trio we can build a histogram over  $R_{enc}$ : The histogram consists of buckets corresponding to the range of all possible values in  $A$ . A histogram bucket with bucket boundary  $[p, q]$  maintains the count of the number of tuples in  $R_{enc}$  that satisfy  $p \leq A < q$ . Now, we can use the bucket frequencies to estimate  $A$ -selectivities by making standard uniformity assumptions.

*Example 5.* For relation  $S_{enc}$  with integer attribute  $A$ , suppose the histogram on  $A$  has bucket intervals 0, 5, 10, and so on; i.e., bucket  $B_{0,5}$  stores the number of tuples in  $S_{enc}$  with  $0 \leq A < 5$ ,  $B_{5,10}$  for  $5 \leq A < 10$ , etc. The number of tuples in  $S_{enc}$  satisfying  $3 \leq A \leq 8$  is estimated as  $\frac{2 \cdot B_{0,5}}{5} + \frac{4 \cdot B_{5,10}}{5}$ . □

### 5.3 X-tuple Counts

For relation  $R$  with attribute  $A$ , we would also like to estimate the number of x-tuples in  $R$  that contain some alternative satisfying a point or range predicate on  $A$ :

- X-Selectivity( $R, A, v$ ): The number of x-tuples in  $R$  containing at least one alternative satisfying  $A = v$ .
- X-Selectivity( $R, A, x, y$ ): The number of x-tuples in  $R$  containing at least one alternative that satisfies  $x \leq A \leq y$ .

*Example 6.* Consider relation  $R$  from Example 3 again. Here X-Selectivity( $R$ , Name, Thomas) = 1 and X-Selectivity( $R$ , Salary, 40K, 60K) = 2. □

**Estimating X-Selectivity( $R, A, v$ ):** Just as for estimating A-selectivities, here too we can build a histogram that now counts the number of x-tuples instead of the number of alternatives. No existing histograms can be used, however, and we build new kinds of histograms for this purpose.

Let us first attempt constructing a histogram by creating buckets on attribute  $A$ , and we shall then improve on this histogram. Within a bucket  $[p, q]$ , we store the number of x-tuples that contain at least one alternative satisfying  $p \leq A < q$ . Note that, unlike in the case of the histogram for A-selectivity whose buckets counted distinct alternatives, in this histogram a single x-tuple may contribute to the count in multiple buckets. For instance, if an x-tuple contains an alternative with  $A$  in range  $[p_1, q_1]$  as well as another alternative with  $A$  in range  $[p_2, q_2]$ , then the x-tuple will contribute to the counts in both the buckets. Given a value  $v$ , suppose  $v$  corresponds to bucket  $[p, q]$ , i.e.,  $p \leq v < q$ , then we estimate X-selectivity( $R, A, v$ ) to be  $\frac{B_{p,q}}{(q-p)}$ , where  $B_{p,q}$  is the stored count in bucket  $[p, q]$ .

Note that the above estimate for a specific value  $v$  assumes that if an x-tuple is in bucket  $[p, q]$ , the probability of it having an alternative with value  $v$ , where  $p \leq v < q$  is  $\frac{1}{(q-p)}$ . However, we can refine this probability if we know the number of alternatives in an x-tuple. Suppose an x-tuple has  $k$  alternatives, whose  $A$  values are randomly and independently distributed between  $p$  and  $q$ , the probability of having at least one alternative with value  $v$  is  $1 - \left(\frac{q-p-1}{q-p}\right)^k$ . (If we know that the distinct  $A$  values in the alternative are independently and randomly distributed in  $[p, q]$ , then we replace  $k$  with the number of distinct  $A$  values in the x-tuple.)

We can use the observation above to enhance our histogram. We construct a 2-D histogram with  $A$  as one dimension, and the size  $\mathcal{S}$  of the x-tuple (either as the number of alternatives or the number of distinct  $A$  values, based on the distribution described above) as the second dimension. In a bucket  $[p, q]$  for dimension  $A$ , and  $[n_1, n_2]$  for dimension  $\mathcal{S}$ , we store the number of x-tuples having either: (1) at least  $k$  alternatives with  $A$  value in the range  $[p, q]$ , or (2) alternatives with at least  $k$  distinct  $A$  values in the range  $[p, q]$ . Recall, we use (1) above if we assume the distribution of all alternatives is independent, and we use (2) if we assume the distribution of distinct  $A$  values is independent.

**Estimating X-Selectivity( $R, A, x, y$ ):** While at first glance it might seem that we can use the histogram described above to estimate X-selectivity( $R, A, x, y$ ) also, that is not the case. We would get an incorrect estimate using the histogram described above because each x-tuple contributes to the count in multiple buckets.<sup>2</sup> Hence if we aggregate counts over multiple buckets, we would end up double-counting x-tuples corresponding to multiple buckets, as shown by the following example.

*Example 7.* Suppose for relation  $S$ , the histogram on attribute  $A$  is bucketed at multiples of 5. If we want an estimate of X-Selectivity( $S, A, 0, 9$ ), adding up the counts corresponding to buckets  $[0, 5]$  and  $[5, 10]$  gives an incorrect answer, because x-tuples could contain both alternatives with  $A$  values in  $[0, 5)$  as well as  $A$  values in  $[5, 10)$ . Hence the returned sum would be an overestimate of the actual count.  $\square$

We need to create a more complex histogram to be able to give accurate expected estimates. We use a 3-D histogram with dimensions  $A_{min}$ ,  $A_{max}$ , and size  $\mathcal{S}$ .  $A_{min}$  corresponds to the minimum  $A$  value for an alternative in an x-tuple,  $A_{max}$  corresponds to the maximum  $A$  value. As before  $\mathcal{S}$  corresponds to the size of the alternatives falling in the bucket, where size is given either by the number of alternatives or by the number of distinct  $A$  values, depending on the distribution assumption. In the histogram bucket corresponding to range  $[p, q]$  for  $A_{min}$ ,  $[r, s]$  for  $A_{max}$ , and  $[n_1, n_2]$  for  $\mathcal{S}$ , we store the number of x-tuples with size  $\mathcal{S}$  in range  $[n_1, n_2]$  that have the minimum  $A$  value in range  $[p, q)$ , maximum  $A$  value in range  $[r, s)$ . Note that now every single x-tuple corresponds to exactly one bucket of the histogram. Hence, we can estimate X-Selectivity( $R, A, x, y$ ) by adding up the contributions from all relevant buckets without any danger of double-counting.

## 5.4 Other Statistics

Finally, we note that several other kinds of statistics can be estimated using the estimation techniques described above. For example, suppose for relation  $R$  we want to know the average number of alternatives satisfying  $A = 5$  among x-tuples that contain at least one alternative with  $A = 5$ , we can compute the average as  $\frac{A\text{-selectivity}(R, A, 5)}{X\text{-selectivity}(R, A, 5)}$ .

Another similar statistic worth mentioning that can be estimated using techniques described earlier is finding the total or average number of alternatives in  $R$  in x-tuples that contain at least one alternative satisfying a predicate:

- A-Average( $R, A$ ): The average number of alternatives in x-tuples that contain at least one alternative satisfying  $A = v$  or  $A \in [x, y]$ .

For A-Average we can use a histogram similar to the one for X-Selectivity, but by eliminating the size dimension. We can then store the total count of alternatives from the x-tuples in each bucket, instead of storing the number of x-tuples.

<sup>2</sup> By ‘incorrect’, we mean that the estimator will not be an unbiased estimator for X-Selectivities, even under standard uniformity assumptions.

## 6 Discussion and Future Work

In this section we discuss important issues arising from this paper that we do not address, as well as more general directions for the future that our work suggests.

**Construction and Maintenance:** The first question that arises about our techniques presented in Section 5 is how we construct the histograms that enable estimating the various statistics. Construction of a histogram entails deciding the bucket boundaries for all the dimensions followed by populating the counts in each bucket. Recall that our histograms for ULDB relations translate to statistics on the encoded conventional relations. Hence we can leverage previously proposed techniques for determining the bucket boundaries and constructing the histogram through sampling [9, 30]. Alternately, we could scan the entire relations apriori and compute exact frequencies for all buckets.

The second issue that arises is that of incrementally maintaining histograms as data is modified. While at first glance it seems like we can still use traditional techniques for maintaining histograms [23], we plan to explore whether there are more efficient techniques that can be applied specific to the kinds of data modifications Trio supports. [13]

**Operators and Indexes:** An interesting followup of our work would be to study whether there are any other specialized operators and indexes that would be useful for query execution in Trio. One possibility we considered is to combine the grouping operator we have proposed with other operators such as various forms of join operators. However, this does not seem to provide any obvious additional benefit over using the join and grouping operator in sequence. (For instance, combining XGB with a hash join might on one hand perform fewer operations, but on the other hand require more memory.) Detailed study of combining groups of operators to form a single more efficient operator is left as future work.

Another issue that we have not addressed is the problem of automatically deciding the indexes to be created given a specific instance of a database and workload of queries, in the spirit of design advisers provided by most commercial DBMSs.

**System:** We intend to implement the new operators and techniques presented in this paper, and incorporate them into the Trio system. We then plan to perform an extensive experimental evaluation quantifying the benefits these techniques add to the current layered approach.

**Statistics Propagation:** We have proposed techniques for estimating statistics for input relations. These statistics can be used in conjunction with conventional sampling-based approaches [2, 15] to estimate certain statistics, alternative and x-tuple cardinalities, for intermediate results output by operators in a query plan. However, reusing other recent techniques such as sketches [20] and wavelets [21, 22] for estimating all statistics of intermediate results to uncertain databases forms an interesting direction of future work.

**Other Models and Systems:** In this paper we've presented techniques that are motivated by the Trio system and its data model. However, some of our techniques can be

adapted for other data models that use similar constructs [3, 5, 12, 17, 28, 32]. More generally, it would be interesting to explore operators and associated estimation techniques for other uncertain database systems. The grand goal of building a generic optimizer for uncertain databases poses several challenging problems including efficient indexing, estimating statistics, constructing and costing query plans, and much more.

## 7 Related Work

The study of uncertainty in databases goes back to the early 80s. A large body of previous work has been theoretical in nature and a very small subset of it can be found in [1, 5, 12, 16, 18, 24–27, 36]. Recently there have been several efforts in building systems for managing uncertainty [4, 8, 10, 29]. While most of these projects have studied efficient algorithms for query processing, none of them actually addresses the problem of query optimization through specialized indexes, operators, and statistics estimation, which are the subject of our paper.

Two notable pieces of recent work studying query processing in probabilistic databases are [11, 14], whose focus is on probability computation. Reference [11] characterizes when a query can be computed using a *safe plan*, which can be very efficiently executed. Reference [14] studies how lineage in Trio can aid in making confidence computation more efficient. Importantly, it shows that lineage allows us to decouple data and confidence computation in Trio, enabling us to use any query plan for data computation. It is this decoupling that allows the optimizer to consider any query plan, without worrying about confidence computation.

Reference [34] proposes interesting new techniques for indexing uncertain data with probabilities. The indexes proposed in [34] are different from ours and focus on the probabilities. Their indexes are very useful for queries with thresholds on probabilities. However, as described above, Trio adopts lineage-based probability computation, and hence we focus only on data computation in this paper. We have proposed new indexing mechanisms specific to the ULDB data model and its relational encoding, which are more useful for Trio query processing. Finally, [34] does not consider the problem of estimating various kinds of statistics on uncertain data.

Obviously there has been a huge body of work studying every aspect of query optimization for conventional databases. Since there have been many papers written on every topic, including relational operations, indexing, histograms, statistics, and query plan selection, we do not review this literature here. We refer the interested reader to any standard database textbook, such as [19].

## 8 Conclusions

We argued that Trio’s current architecture of layering on top of a conventional DBMS can be enhanced for better performance by exploiting the the uniformity in the structure of queries and encoded Trio relations. To this end, we proposed novel indexing techniques in Trio, and defined a new interesting order, which can be useful in assembling query plans. We devised a relational operator that ensures the interesting order on its output. To guide the query optimizer in choosing the optimal query plan, we designed

histograms that enable estimating various useful statistical properties of uncertain data. Finally, we suggested several avenues for interesting future work in the area of query optimization for Trio and other uncertain database systems.

## References

1. S. Abiteboul, P. Kanellakis, and G. Grahne. On the Representation and Querying of Sets of Possible Worlds. *Theoretical Computer Science*, 78(1), 1991.
2. N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proc. of ACM PODS*, 1999.
3. P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.
4. L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *Proc. of ICDE*, 2007.
5. D. Barbará, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Trans. Knowl. Data Eng.*, 4(5), 1992.
6. O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of VLDB*, 2006.
7. O. Benjelloun, A. Das Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB Journal*, 17(2), 2008.
8. J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. of ACM SIGMOD*, 2005.
9. S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: how much is enough? In *Proc. of ACM SIGMOD*, 1998.
10. R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *Proc. of VLDB*, 2005.
11. N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *Proc. of VLDB*, 2004.
12. A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working Models for Uncertain Data. In *Proc. of ICDE*, 2006.
13. A. Das Sarma, M. Theobald, and J. Widom. Data modifications and versioning in trio. Technical report, Stanford InfoLab, 2008. Available at <http://dbpubs.stanford.edu/pub/2008-5>.
14. A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proc. of ICDE*, 2008.
15. C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *Proc. of ICDE*, 2006.
16. N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. In *Proc. of VLDB*, 1990.
17. N. Fuhr and T. Rölleke. A Probabilistic NF2 Relational Algebra for Imprecision in Databases. *Unpublished Manuscript*, 1997.
18. N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM TOIS*, 14(1), 1997.
19. H. G-Molina, J. Widom, and J. D. Ullman. *Database Systems: The Complete Book*. Prentice-Hall, 2002.
20. S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *Proc. of EDBT*, 2004.
21. M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proc. of ACM SIGMOD*, 2002.

22. M. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proc. of ACM PODS*, 2004.
23. P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. of VLDB*, 1997.
24. G. Grahne. Dependency Satisfaction in Databases with Incomplete Information. In *Proc. of VLDB*, 1984.
25. G. Grahne. Horn Tables - An Efficient Tool for Handling Incomplete Information in Databases. In *Proc. of ACM PODS*, 1989.
26. T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *Proc. of IIDB Workshop*, 2006.
27. T. Imielinski and W. Lipski Jr. Incomplete Information in Relational Databases. *Journal of the ACM*, 31(4), 1984.
28. S. K. Lee. An extended Relational Database Model for Uncertain and Imprecise Information. In *Proc. of VLDB*, 1992.
29. M. Mutsuzaki, M. Theobald, A. Keijer, J. Widom, P. Agrawal, O. Benjelloun, A. Das Sarma, R. Murthy, and T. Sugihara. Trio-one: Layering uncertainty and lineage on a conventional DBMS. In *Proc. of CIDR*, 2007. Demonstration description.
30. G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of ACM SIGMOD*, 1984.
31. C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. of ICDE*, 2007.
32. C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *VLDB*, 2007.
33. P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Proc. of ICDE*, 2007.
34. S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *Proc. of ICDE*, 2007.
35. J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proc. of CIDR*, 2005.
36. J. Wijsen. Condensed representation of database repairs for consistent query answering. In *Proc. of ICDT*, 2003.