Mimir: Bringing CTables into Practice*

Arindam Nandi[β], Ying Yang[β], Oliver Kennedy[β]
Boris Glavic[i], Ronny Fehling[α], Zhen Hua Liu[o], Dieter Gawlick[o]

University at Buffalo[β] Illinois Institute of Technology[i] Airbus[α] Oracle[o]

{arindamn, yyang25, okennedy}@buffalo.edu bglavic@iit.edu
ronny.fehling@airbus.com {zhen.liu,dieter.gawlick}@oracle.com

ABSTRACT

The present state of the art in analytics requires high upfront investment of human effort and computational resources to curate datasets, even before the first query is posed. So-called pay-as-you-go data curation techniques allow these high costs to be spread out, first by enabling queries over uncertain and incomplete data, and then by assessing the quality of the query results. We describe the design of a system, called Mimir, around a recently introduced class of probabilistic pay-as-you-go data cleaning operators called Lenses. Mimir wraps around any deterministic database engine using JDBC, extending it with support for probabilistic query processing. Queries processed through Mimir produce uncertainty-annotated result cursors that allow client applications to quickly assess result quality and provenance. We also present a GUI that provides analysts with an interactive tool for exploring the uncertainty exposed by the system. Finally, we present optimizations that make Lenses scalable, and validate this claim through experimental evidence.

Keywords

Uncertain Data, Provenance, ETL, Data Cleaning

1. INTRODUCTION

Data curation is presently performed independently of an analyst's needs. To trust query results, an analyst first needs to establish trust in her data, and this process typically requires high upfront investment of human and computational effort. However, the level of cleaning effort is often not commensurate with the specific analysis to be performed. A class of so-called pay-as-you-go [19], or on-demand [36] data cleaning systems have arisen to flatten out this upfront cost. In on-demand cleaning settings, an analyst quickly applies data cleaning heuristics without needing to tune the process or supervise the output. As the analyst poses queries, the on-demand system continually provides feedback about the quality and precision of the query results. If the analyst wishes higher quality, more precise results, the system can also provide guidance to focus the analyst's data cleaning efforts on curating inputs that are relevant to the analysis.

*The first two authors contributed equally and should be considered a joint first author

Product

id	name	brand	cat	ROWID
P123	Apple 6s, White	?	phone	R1
P124	Apple 5s, Black	?	phone	R2
P125	Samsung Note2	Samsung	phone	R3
P2345	Sony to inches	?	?	R4
P34234	Dell, Intel 4 core	Dell	laptop	R5
P34235	HP, AMD 2 core	HP	laptop	R6

Ratings1

pid	...	rating	review_ct	ROWID
P123	...	4.5	50	R7
P2345	...	?	245	R8
P124	...	4	100	R9

Ratings2

pid	...	evaluation	num_ratings	ROWID
P125	...	3	121	R10
P34234	...	5	5	R11
P34235	...	4.5	4	R12

Figure 1: Incomplete error-filled example relations, including an implicit unique identifier attribute ROWID.

In this paper we describe Mimir, a system that extends existing relational database engines with support for on-demand curation. Mimir is based on **lenses** [36], a powerful and flexible new primitive for on-demand curation. Lenses promise to enable a new kind of uncertainty-aware data analysis that requires minimal up-front effort from analysts, without sacrificing trust in the results of that analysis. Mimir is presently compatible with SQLite and a popular commercial enterprise database management system.

In the work that first introduced Lenses [36] we demonstrated how curation tasks including *domain constraint repair*, *schema matching*, and *data archival* can be expressed as lenses. Lenses in general, are a family of unary operators that (1) Apply a data curation heuristic to clean or validate their input, and (2) Annotate their output with all assumptions or guesses made by the heuristic. Critically, lenses require little to no *upfront* configuration — the lens' output represents a best-effort guess. Previous efforts on uncertain data management [12] focus on producing exclusively correct, or *certain* results. By comparison, lenses *may* include incorrect results. Annotations on the lens output persist through queries and provide a form of provenance that helps analysts understand potential sources of error and their impact on query results. This in turn allows an analyst to decide whether or not to trust query results, and how to best allocate limited resources to data curation efforts.

Example 1. *Alice is an analyst at a retail store and is developing a promotional strategy based on public opinion ratings gathered by two data collection companies. A thor-*

ough analysis of the data requires substantial data curation effort from Alice: As shown in Figure 1, the rating company's schemas are incompatible, and the store's own product data is incomplete. However, Alice's preliminary analysis is purely exploratory, and she is hesitant to invest the effort required to fully curate this data. She creates a lens to fix missing values in the Product table:

```
CREATE LENS SaneProduct AS SELECT * FROM Product
USING DOMAIN_REPAIR(cat string NOT NULL,
                    brand string NOT NULL);
```

From Alice's perspective, the lens `SaneProduct` *behaves as a standard database view. However, the content of the lens is guaranteed to satisfy the domain constraints on* `category` *and* `brand`*. NULL values in these columns are replaced according to a classifier built over the* `Product` *table. Under the hood, the Mimir system maintains a probabilistic version of the view as a so-called Virtual C-Table (VC-Table). A VC-Table cleanly separates the* ***existence*** *of uncertainty (e.g., the category value of a tuple is unknown), the* ***explanation*** *for how the uncertainty affects a query result (this is a specific type of provenance), and the* ***model*** *for this uncertainty as a probability distribution (e.g., a classifier for category values that is built when the lens is created).*

Uncertainty is encoded in a VC-Table through attribute values that are symbolic expressions over variables representing unknowns. A probabilistic model for these variables is maintained separately. Queries over a VC-Table can be translated into deterministic SQL over the lens' deterministic inputs. This is achieved by evaluating deterministic expressions as usual and by manipulating symbolic expressions for computations that involve variables. The result is again a relational encoding of a VC-Table. The probabilistic model (or an approximation thereof) can be "plugged" into the expressions in a post-processing step to get a deterministic result. This approach has several advantages: (1) the probabilistic model can be created in a pay-as-you-go fashion focusing efforts on the part that is relevant for an analyst's query; (2) the symbolic expressions of a VC-Table serve as a type of provenance that explain how the uncertainty affects the query result; (3) changes to the probabilistic model or switching between different approximations for a model only require repetition of the post-processing step over the already computed symbolic query result; (4) large parts of the computation can be outsourced to a classical relational database; and (5) queries over a mixture of VC-Tables and deterministic tables are supported out of the box. A limitation of our preliminary work with VC-Tables is the scalability of the expressions outsourced to the deterministic database. Our initial approach sometimes creates outsourced expressions that can not be evaluated efficiently. In this paper, we address this limitation, and in doing so demonstrate that VC-Tables are a scalable and practical tool for managing uncertain data. Concretely, in this paper:

- In Section 3, we describe the Mimir system, including its APIs, its user interface, and a novel "annotated" result cursor that enables uncertainty-aware analytics.
- In Section 4, we demonstrate the limitations of naive VC-Tables and introduce techniques for scalable query processing over VC-Tables.
- In Section 5, we evaluate Mimir on SQLite and a commercial database system.

$$
\begin{aligned}
e &:= \mathbb{R} \mid \mathit{Column} \mid \textbf{if } \phi \textbf{ then } e \textbf{ else } e \\
&\quad \mid e \ \{+,-,\times,\div\} \ e \mid \mathit{Var}(\mathit{id}[, e[, e[, \ldots]]]) \\
\phi &:= e \ \{=,\neq,<,\leq,>,\geq\} \ e \mid \phi \ \{\wedge,\vee\} \ \phi \mid \top \mid \bot \\
&\quad \mid e \texttt{ is null} \mid \neg\phi
\end{aligned}
$$

Figure 2: Grammars for boolean expressions ϕ and numerical expressions e including VG-Functions $Var(\ldots)$.

2. BACKGROUND

Possible Worlds Semantics. An uncertain database $\mathcal{D}$ over a schema $sch(\mathcal{D})$ is defined as a set of possible worlds: deterministic database instances $D \in \mathcal{D}$ over schema $sch(D) = sch(\mathcal{D})$. Possible worlds semantics defines queries over uncertain databases in terms of deterministic query semantics. A deterministic query Q applied to an uncertain database defines a set of possible results $Q(\mathcal{D}) = \{\ Q(D) \mid D \in \mathcal{D}\ \}$. Note that these semantics are agnostic to the data representation, query language, and number of possible worlds $|\mathcal{D}|$. A *probabilistic database* $\langle\ \mathcal{D}, p\ \rangle$ is an uncertain database annotated with a probability distribution $p : \mathcal{D} \to [0,1]$ that induces a distribution over all possible result relations R:

$$P[Q(\mathcal{D}) = R] = \sum_{D \in \mathcal{D} \,:\, Q(D)=R} p(D)$$

A probabilistic query processing (PQP) system is supposed to answer a deterministic query Q by listing all its possible answers and annotating each tuple with its marginal probability. These tasks are often #P-hard in practice, necessitating the use of approximation techniques.

C-Tables and PC-Tables. One way to make probabilistic query processing efficient is to encode $\mathcal{D}$ and P through a compact, factorized representation. In this paper we adopt a generalized form of C-Tables [17, 22] to represent $\mathcal{D}$, and PC-Tables [15, 21] to represent the pair $(\mathcal{D}, P)$. A C-Table [17] is a relation instance where each tuple is annotated with a formula ϕ, a propositional formula over an alphabet of variable symbols Σ. The formula ϕ is often called a *local condition* and the symbols in Σ are referred to as *labeled nulls*, or just variables. Intuitively, for each assignment to the variables in Σ we obtain a possible relation containing all the tuples whose formula ϕ is satisfied. For example:

	Product				
	pid	**name**	**brand**	**category**	ϕ
t_1	P123	Apple 6s	Apple	phone	$x_1 = 1$
t_2	P123	Apple 6s	Cupertino	phone	$x_1 = 2$
t_3	P125	Note2	Samsung	phone	$\top$

$$x_1 = \begin{cases} 1 : 0.3 \\ 2 : 0.7 \end{cases}$$

The above C-Table defines a set of two possible worlds, $\{t_1, t_3\}$, $\{t_2, t_3\}$, i.e. one world for each possible assignment to the variables in the one-symbol alphabet $\Sigma = \{x_1\}$. Notice that no possible world can have both t_1 and t_2 at the same time. Adding a probabilistic model for the variables, e.g., $P(x_1)$ as shown above, we get a PC-table. For instance, in this example the probability that the brand of product $P123$ is *Apple* is 0.3. C-Tables are closed w.r.t. positive relational algebra [17] : if $\mathcal{D}$ is representable by a C-Table and Q is a positive query then $\mathcal{D}' = \{\ Q(D) \mid D \in \mathcal{D}\ \}$ is representable by another C-Table.

VG-Relational Algebra. VG-RA (variable-generating relational algebra) [22] is a generalization of positive bag-relation algebra with extended projection, that uses a simplified form of VG-functions [18]. In VG-RA, VG-functions (i) dynamically introduce new Skolem symbols in Σ, that

are guaranteed to be unique and deterministically derived by the function's parameters, and (ii) associate the new symbols with probability distributions. Hence, VG-RA can be used to define new PC-Tables. Primitive-valued expressions in VG-RA (i.e., projection expressions and selection predicates) use the grammar summarized in Figure 2. The primary addition of this grammar is the VG-Function term representing unknown values: $Var(\ldots)$.

VG-RA's expression language enables a generalized form of C-Tables, where attribute-level uncertainty is encoded by replacing missing values with VG-RA *expressions* (not just variables) that act as freshly defined Skolem terms. For example, the previous PC-Table is equivalent to the generalized PC-Table:

Product			
pid	**name**	**brand**	**category**
P123	Apple 6s	$Var('X', \texttt{R1})$	phone
P125	Note2	Samsung	phone

$$Var('X', \texttt{R1}) = \begin{cases} Apple : 0.3 \\ Cupertino : 0.7 \end{cases}$$

It has been shown that generalized C-Tables are closed w.r.t VG-RA [17, 22]. Evaluation rules for VG-RA use a *lazy evaluation* operator $[[\cdot]]_{lazy}$, which uses a *partial* binding of $Column$ or $Var(\ldots)$ atoms to corresponding expressions. Lazy evaluation applies the partial binding and then reduces every sub-tree in the expression that can be deterministically evaluated. Non-deterministic sub-trees are left intact.

Any tuple attribute appearing in a C-Table can be encoded as an abstract syntax tree for a partially evaluated expression that assigns it a value. This is the basis for evaluating projection operators, where every expression e_i in the projection's target list is lazily evaluated. Column bindings are given by each tuple in the source relation. The local condition ϕ is preserved intact through the projection. Selection is evaluated by combining the selection predicate ϕ with each tuple's existing local condition. For example, consider a query $\pi_{brand,category}(\sigma_{brand=Apple}(Product))$ over the example PC-Table. The result of this query is shown below. The second tuple of the input table does not fulfil the selection condition and is thus guaranteed to not be in the result. Note the symbolic expressions in the local condition and attribute values. Furthermore, note that the probabilistic model for the single variable is not influenced by the query at all.

Query Result		
brand	**category**	ϕ
$Var('X', \texttt{R1})$	phone	$Var('X', \texttt{R1}) = Apple$

From now on, we will implicitly assume this generalized form of C-Tables.

Lenses. Lenses use VG-RA queries to define new C-Tables as views: A lens defines an uncertain view relation through a VG-RA query $\mathcal{F}_{lens}(Q(D))$, where $\mathcal{F}$ and Q to represents the non-deterministic an deterministic components of the query, respectively. Independently, the lens constructs P as a joint probability distribution over every variable introduced by $\mathcal{F}_{lens}$, by defining a sampling process in the style of classical VG-functions [18], or supplementing it with additional meta-data to create a PIP-style grey-box [22]. These semantics are closed over PC-Tables. If $Q(D)$ is non-deterministic — that is, the lens' input is defined by a PC-Table $(Q(D), P_Q)$ — the lens' semantics are virtually unchanged due to the closure of VG-RA over C-Tables.

Example 2. *Recall the lens definition from Example 1. This lens defines a new C-Table using the VG-RA query:*

$$\pi_{id \leftarrow id, name \leftarrow name, brand \leftarrow f(brand), cat \leftarrow f(cat)}(Product)$$

In this expression f denotes a check for domain compliance, and a replacement with a non-deterministic value if the check fails, as follows:

$$f(x) \equiv \textbf{if } x \texttt{ is null } \textbf{then } Var(x, \texttt{ROWID}) \textbf{ else } x$$

The models for $Var('brand', \texttt{ROWID})$ and $Var('cat', \texttt{ROWID})$ are defined by classifiers trained on the contents of $Product$.

Virtual C-Tables. Consider a probabilistic database in which all non-determinism is derived from lenses. In this database, all C-Tables, including those resulting from deterministic queries over non-deterministic data can be expressed as VG-RA queries over a deterministic database D. Furthermore, VG-RA admits a normal form [36] for queries where queries are segmented into a purely deterministic component $Q(D)$ and a non-deterministic component $\mathcal{F}(Q(D))$. These normalization rules are shown in Figure 3.

Normalization does not affect the linkage between the C-Table computed by a VG-RA query and its associated probability measure P: $Var(\ldots)$ remains unchanged. Moreover, the non-deterministic component of the normal form $\mathcal{F}$ is a simple composite projection and selection operation.

The simplicity of $\mathcal{F}$ carries two benefits. First, the deterministic component of the query can be evaluated natively in a database engine, while the non-deterministic component can be applied through a simple shim interface wrapping around the database. Second, the abstract syntax tree of the expression acts a form of provenance [2, 3] that annotates uncertain query results with metadata describing the level and nature of their uncertainty, a key component of the system we now describe. For example, in the query result shown above it is evident that the tuple will be in the result as long as the condition $Var('X', \texttt{R1}) =$ 'Apple' evaluates to true. Mimir provides an API for the user to retrieve this type of explanation for a query result and comes with a user interface that visualizes explanations.

3. SYSTEM OUTLINE

The Mimir system is a shim layer that wraps around an existing DBMS to provide support for lenses. Using Mimir, users define lenses that perform common data cleaning operations such as schema matching, missing value interpolation, or type inference with little or no configuration on the user's part. Mimir exports a native SQL query interface that allows lenses to be queried as if they were ordinary relations in the backend database. A key design feature of Mimir is that it has minimal impact on its environment. Apart from using the backend database to persist metadata, Mimir does not modify the database or its data in any way. As a consequence, Mimir can be used alongside any existing database workflow with minimal effort and minimal risk.

3.1 User Interface

Users define lenses through a `CREATE LENS` statement that immediately instantiates a new lens.

Example 3. *Recall the example data from Figure 1. To merge the two ratings relations, Alice needs to re-map the attributes of `Ratings2`. Rather than doing so manually, she defines a lens that re-maps the attributes of the `Ratings2` relation to those of `Ratings1` as follows.*

$$\pi_{a'_j \leftarrow e'_j}\left(\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D))\right) \equiv \mathcal{F}(\langle\, a'_j \leftarrow [[e'_j(a_i \leftarrow e_i)]]_{lazy} \,\rangle, \phi)(Q(D)) \tag{1}$$

$$\sigma_{\psi}\left(\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D))\right) \equiv \mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi \wedge \psi_{var})(\sigma_{\psi_{det}}(Q(D))) \tag{2}$$

$$\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D)) \times \mathcal{F}(\langle\, a'_j \leftarrow e'_j \,\rangle, \phi')(Q'(D)) \equiv \mathcal{F}(\langle\, a_i \leftarrow e_i, a'_j \leftarrow e'_j \,\rangle, \phi \wedge \phi')(Q(D) \times Q'(D)) \tag{3}$$

$$\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D)) \uplus \mathcal{F}(\langle\, a_i \leftarrow e'_i \,\rangle, \phi')(Q'(D)) \equiv$$
$$\mathcal{F}(\langle\, a_i \leftarrow [[\textbf{if } src = 1 \textbf{ then } e_i \textbf{ else } e'_i]]_{lazy} \,\rangle, [[\textbf{if } src = 1 \textbf{ then } \phi \textbf{ else } \phi']]_{lazy})(\pi_{*,src \leftarrow 1}(Q(D)) \uplus \pi_{*,src \leftarrow 2}(Q'(D))) \tag{4}$$

Figure 3: Reduction to VG-RA Normal Form.

```
CREATE LENS MatchedRatings2 AS
SELECT * FROM Ratings2
USING SCHEMA_MATCHING(pid string, ...,
      rating float, review_ct float, NO LIMIT);
```

CREATE LENS statements behave like a view definition, but also apply a data curation step to the output; in this case schema matching. Mapping targets may be defined explicitly or by selecting an existing relation's schema in the GUI.

In addition to a command-line tool, Mimir provides a Graphical User Interface (GUI) illustrated in Figure 4. Users pose queries over lenses and deterministic relations using standard SQL `SELECT` statements (a). Mimir responds to queries over lenses with a *best guess* result, or the result of the query in the possible world with maximum likelihood. In contrast to the classical notion of "certain" answers, the best guess *may* contain inaccuracies. However, all uncertainty arises from Var terms introduced by lenses. Consequently, using the provenance of each row and cell, Mimir can identify potential sources of error: Attribute values that depend on a Var term may be incorrect, and filtering predicates that depend on a Var term may lead to rows incorrectly being included or excluded in the result. We refer to these two types of error as *non-deterministic cells*, and *non-deterministic rows*, respectively.

Example 4. *Recall the result of the example query in Section 2, which shows a VC-table before the best guess values are plugged in. The only row is non-deterministic, because its existence depends on the value of* $Var('X', R1)$ *which denotes the unknown brand of this tuple. The brand attribute value of this tuple is a non-deterministic cell, because its value depends on the same expression.*

In Mimir, query results (b) visually convey potential sources of error through several simple cues. First, a small provenance graph (c) helps the user quickly identify the data's origin, what cleaning heuristics have been applied, and where.

Potentially erroneous results are clearly identified: Non-deterministic rows have a red marker on the right and a grey background, while non-deterministic cells are highlighted in red.

Clicking on a non-deterministic row or cell brings up an explanation window (d). Here, Mimir provides the user with statistical metrics summarizing the uncertainty of the result, as well as a list of human-readable reasons why the result might be incorrect. Each reason is linked to a specific lens; If the user believes a reason to be incorrect, she can click "Fix" to override the lens' data cleaning decision. An "Approve" button allows a user to indicate that the lens heuristic's choice is satisfactory. Once all reasons for a given row or value's non-determinism have been either approved or fixed,

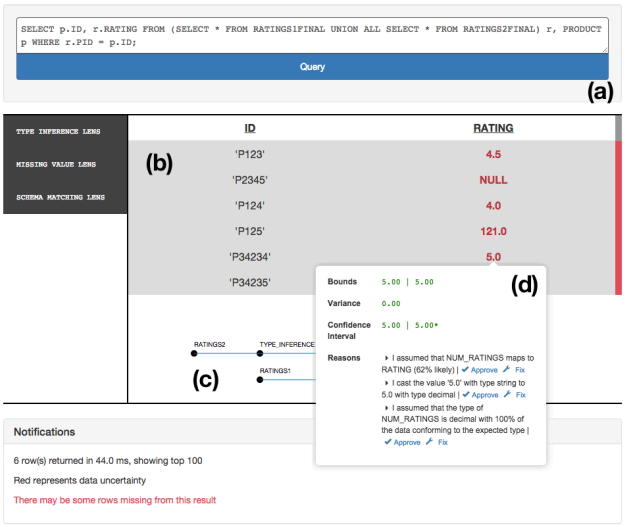


Figure 4: The Graphical Mimir User Interface

the row or value becomes green to signify that it is now deterministic.

Example 5. *Figure 4 shows the results of a query where one product (with id 'P125') has an unusually high rating of 121.0. By clicking on it, Alice finds that a schema matching lens has incorrectly mapped the* `NUM_RATINGS` *column of one input relation to the* `RATINGS` *column of the other input relation — 121 is the number of ratings for the product, not the actual rating itself. By clicking on the fix button, Alice can manually specify the correct match and Mimir re-runs the query with the correct mapping.*

Making sources of uncertainty easily accessible allows the user to quickly track down errors that arise during heuristic data cleaning, even while viewing the results of complex queries. Limiting Mimir to simple signifiers like highlighting and notifications prevents the user from being overwhelmed by details, while explanation windows still allow the user to explore uncertainty sources in more depth at their own pace.

3.2 The Mimir API

The Mimir system's architecture is shown in Figure 5. Mimir acts as an intermediary between users and a back-end database using JDBC. Mimir exposes the database's native SQL interface, and extends it with support for lenses. The central feature of this support is five new functions in the JDBC result cursor class that permit client applications such as the Mimir GUI to evaluate result quality. The first three indicate the presence of specific classes of un-

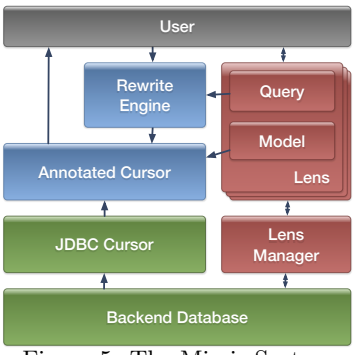


Figure 5: The Mimir System

certainty: (1) `isColumnDeterministic(int | String)` returns a boolean that indicates whether the value of the indicated attribute was computed deterministically without having to "plug in" values for variables. In our graphical interface, cells for which this function returns false are highlighted in red. Note that the same column may contain both deterministic and non-deterministic values (e.g., for a lens that replaces missing values with interpolated estimates) (2) `isRowDeterministic()` returns a boolean that indicates whether the current row's presence in the output can be determined without using the probabilistic model. In our graphical interface, rows for which this function returns false are also highlighted. (3) `nonDeterministicRowsMissing()` returns a count of the number of rows that have been so far ommitted from the result, and were discarded based on the output of a lens. In our graphical interface, when this method returns a number greater than zero after the cursor is exhausted, a notification is shown on the screen.

As we discuss below, limiting the response of these functions to a simple boolean makes it possible to evaluate them rapidly, in-line with the query itself. For additional feedback, Mimir provides two methods: `explainColumn(int | String)` and `explainRow()` Both methods construct and return an explanation object as detailed below. In the graphical Mimir interface, these methods are invoked when a user clicks on a non-deterministic (i.e., highlighted) row or cell, and the resulting explanation object is used to construct the uncertainty summary shown in the explanation window. Explanations do not need to be computed in-line with the rest of the query, but to maintain user engagement, explanations for individual rows or cells still need to be computed quickly when requested.

3.3 Lens Models

A lens consists of two components: (1) A VG-RA expression that computes the output of the lens, introducing new variables in the process using Var terms, and (2) A model object that defines a probability space for every introduced variable. Recall that Var terms act as skolem functions, introducing new variable symbols based on their arguments. For example the `ROWID` attribute can be used to create a distinct variable named "X" for every row using the expression $Var('X', \texttt{ROWID})$. Correspondingly, we distinguish between Var terms and the variable instances they create. Note that the latter is uniquely identified by the name and arguments of the Var term. The model object has three mandatory methods: (1) `getBestGuess(var)` returns the value of the specified variable instance in the most likely possible world. (2) `getSample(var, id)` returns the value of the specified variable instance in a randomly selected possible world. id acts as a seed value, ensuring that the same possible world is selected across multiple calls. (3) `getReason(var)` returns a human-readable explanation of the heuristic guess represented by the specified variable instance. The `getBestGuess` method is used to produce best-guess query results. The remaining two methods are used by explanation objects. As in PIP [22] and Orion 2.0 [31], optional metadata can sometimes permit the use of closed-form solutions when computing statistical metrics for result values.

3.4 Explanation Objects

Explanation objects provide a means for client applications like the GUI to programmatically analyze the non-determinism of a specific row or cell. Concretely, an explanation object provides methods that compute: (1) Statistical metrics that quantitatively summarize the distribution of possible result outcomes, and (2) Qualitative summaries or depictions of non-determinism in the result.

3.4.1 Statistical Metrics

Available statistical metrics depend on whether the explanation object is constructed for a row or a cell, and in the latter case also on what type of value is contained in the cell. For rows, the explanation object has only one method that computes the row's confidence, or the probability that the row is part of the result set.

For numerical cells, the explanation object has methods for computing the value's variance, confidence intervals, and may also be able to provide upper and lower bounds. Variance and confidence intervals are computed analytically if possible, or Monte Carlo style by generating samples of the result using the `getSample` method on all involved models. When computing these metrics using Monte Carlo, we discard samples that do not satisfy the local condition of the cell's row, as the cell will not appear in the result at all if the local condition is false. Statistical metrics are not computed for non-numerical cells.

3.4.2 Qualitative Summaries

Quantitative statistical metrics do not always provide the correct intuition about a result value's quality. In addition to the above metrics, an explanation object can also use Monte Carlo sampling to construct histograms and example result values for non-deterministic cells.

Furthermore, for both cells and rows, an explanation object can produce a list of *reasons* — the human readable summaries obtained from each participating model's `getReason` method. Reasons are ranked according to the relative contribution of each Var term to the uncertainty of the result using a heuristic called CPI [36].

3.5 Query Processing

Queries issued to Mimir are parsed into an intermediate representation (IR) based on VG-RA. Mimir maintains a list of all active lenses as a relation in the backend database. References to lenses in the IR are replaced with the VG-RA expression that defines the lens' contents. The resulting expression is a VG-RA expression over deterministic data residing in the backend database.

Before queries over lenses are evaluated, the query's VG-RA expression is first normalized into the form $\mathcal{F}(Q(D))$,

where $\mathcal{F}(R) = \pi_{a_i \leftarrow e_i}(\sigma_\phi(R))$, ϕ represents a non-deterministic boolean expression, and the subsequent projection assigns the result of non-deterministic expressions e_i to the corresponding attributes a_i. Mimir obtains a classical JDBC cursor for $Q(D)$ from the backend database and constructs an extended cursor using $\mathcal{F}$ and the JDBC cursor.

Recall non-determinism in ϕ and e_i arises from Var terms. When evaluating these expressions, Mimir obtains a specific value for the term using the `getBestGuess` method on the model object associated with each term. Apart from this, ϕ and the e_i expressions are evaluated as normal.

Determining whether an expression is deterministic or not requires slightly more effort. In principle, we could say that any expression is non-deterministic if it contains a Var term. However, it is still possible for the expression's result to be entirely agnostic to the value of the Var.

Example 6. *Consider the following expression, which is used in Mimir's domain constraint repair lens:*

$$\textbf{if } A \texttt{ is null } \textbf{then } Var('A', \texttt{ROWID}) \textbf{ else } A$$

Here, the value of the expression is only non-deterministic for rows where A is null.

Concretely, there are three such cases: (1) conditional expressions where the condition is deterministic, (2) `AND` expressions where one clause is deterministically false, and (3) `OR` expressions where one clause is deterministically true. Observe that these cases mirror semantics for `NULL` and `UNKNOWN` values in deterministic SQL.

For each e_i and ϕ, the Mimir compiler uses a recursive descent through the expression illustrated in Algorithm 1 to obtain a boolean formula that determines whether the expression is deterministic for the current row. These formulas permit quick responses to the `isColumnDeterministic` and `isRowDeterministic` methods. The counter for the `nonDeterministicRowsMissing` method is computed by using `isRowDeterministic` on each discarded row.

Algorithm 1 `isDet`(E)

In: E: An expression in either grammar from Fig. 2.
Out: An expression that is true when E is deterministic.

if $E \in \{\mathbb{R}, \top, \bot\}$ **then**
  return $\top$
else if E **is** Var **then**
  return $\bot$
else if E **is** $Column_i$ **then**
  return $\top$
else if E **is** $\neg E_1$ **then**
  return $\texttt{isDet}(E_1)$
else if E **is** $E_1 \vee E_2$ **then**
  return $(E_1 \wedge \texttt{isDet}(E_1)) \vee (E_2 \wedge \texttt{isDet}(E_2)) \vee (\texttt{isDet}(E_1) \wedge \texttt{isDet}(E_2))$
else if E **is** $E_1 \wedge E_2$ **then**
  return $(\neg E_1 \wedge \texttt{isDet}(E_1)) \vee (\neg E_2 \wedge \texttt{isDet}(E_2)) \vee (\texttt{isDet}(E_1) \wedge \texttt{isDet}(E_2))$
else if E **is** $E_1 \{+, -, \times, \div, =, \neq, >, \geq, <, \leq\} E_2$ **then**
  return $(\texttt{isDet}(E_1) \wedge \texttt{isDet}(E_2))$
else if E **is if** E_1 **then** E_2 **else** E_3 **then**
  return $\texttt{isDet}(E_1) \wedge ((E_1 \wedge \texttt{isDet}(E_2)) \vee (\neg E_1 \wedge \texttt{isDet}(E_3)))$

When one of the explain methods is called, Mimir extracts all of the Var terms from the corresponding expression, and uses the associated model object's `getReason` method to obtain a list of reasons. Variance, confidence bounds, and row-level confidence are computed by sampling from the possible worlds of the model using `getSample` and evaluating the expression in each possible world. Upper and lower bounds are obtained if possible from an optional method on the model object, and propagated through expressions where possible.

4. OPTIMIZING VIRTUAL C-TABLES

The primary scalability challenge that we address in this paper relates to how queries are normalized in Virtual C-Tables. Concretely, the problem arises in the rule for normalizing selection predicates:

$$\sigma_\psi(\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D))) \equiv \mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi \wedge \psi_{var})(\sigma_{\psi_{det}}(Q(D)))$$

Non-deterministic predicates are always pushed into $\mathcal{F}$, including those that could otherwise be used as join predicates. When this happens, the backend database is given a cross-product query to evaluate, and the join is evaluated far less efficiently as a selection predicate in the Mimir shim layer.

In this section, we explore variations on the theme of query normalization. These alternative evaluation strategies make it possible for a traditional database to scalably evaluate C-Table queries, while retaining the functionality of Mimir's uncertainty-annotated cursors as described in Section 3.2. Supporting C-Tables and annotated cursors carries several challenges:

Var Terms. Classical databases are not capable of managing non-determinism, making Var terms functionally into black-boxes. Although a single best-guess value does exist for each term, the models that compute this value normally reside outside of the database.

`isDeterministic` methods. Mimir's annotated cursors must be able to determine whether a given row's presence (resp., a cell's value) depends on any Var terms. Using $\mathcal{F}$, this is trivial, as all Var terms are conveniently located in a single expression that is used to determine the row's presence (resp., to compute a cell's value). Because these methods are used to construct the initial response shown to the user (i.e., to determine highlighting), they must be fast.

Potentially Missing Rows. Annotated cursors must also be able to evaluate the number of rows that could potentially be missing, depending on how the non-determinism is resolved. Although the result of this method is presented to the user as part of the initial query, the value is shown in a notification box and is off of the critical path of displaying the best guess results themselves.

Explanations. The final feature that annotated cursors are expected to support is the creation of explanation objects. These do not need to be created until explicitly requested by the user; the initial database query does not need to be directly involved in their construction. However, it must still be possible to construct and return an explanation object quickly to maintain user engagement.

We now discuss two complimentary techniques for constructing annotated iterators over Virtual C-Tables. Our first approach *partitions* queries into deterministic and non-deterministic fragments to be evaluated separately. The second approach pre-materializes best-guess values into the backend database, allowing it to evaluate the non-deterministic query with Var terms *inlined*.

4.1 Approach 1: Partition

We observe that uncertain data is frequently the minority of the raw data. Moreover, for some lenses, whether a row is deterministic or not is data-dependent. Our first approach makes better use of the backend database by partitioning the query into one or more deterministic and non-deterministic segments, computing each independently, and unioning the results together. When the row-determinism ϕ of a result depends on deterministic data we can push more work into the backend database for those rows that we know to be deterministic. For this deterministic partition of the data, joins can be evaluated correctly and other selection predicates can be satisfied using indexes over the base data. As a further benefit, tuples in each partition share a common lineage, allowing substantial re-use of annotated cursor metadata for all tuples returned by the query on a single partition. To partition a query $\mathcal{F}(Q(D))$, we begin with a set of partitions, each defined by a boolean formula ψ_i over attributes in $sch(Q)$. The set of partitions must be complete ($\bigvee \psi_i \equiv \top$) and disjoint ($\forall i \neq j \;.\; \psi_i \rightarrow \neg\psi_j$). In general, partition formulas are selected such that $\sigma_{\psi_i}(Q(D))$ never contains query results that can be deterministically excluded from $\mathcal{F}(Q(D))$.

Example 7. *Recall the* `SaneProduct` *lens from Examples 1 and 2. Alice the analyst now posses a query:*

```
SELECT name FROM SaneProduct
WHERE brand = 'Apple' AND cat = 'phone'
```

Some rows of the resulting relation are non-deterministic, but only when the `brand` *or* `cat` *in the corresponding row of* `Product` *is* `NULL`*. Optimizing further, all products that are known to be either non-phones or non-Apple products are also deterministically not in the result.*

Given a set of partitions $\Psi = \{\psi_1, \ldots, \psi_N\}$, the partition rewrite transforms the original query into an equivalent set of partitioned queries as follows:

$$(\mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi)(Q(D))) \mapsto \mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi_{var,1})(\sigma_{\psi_1 \wedge \phi_{det,1}}(Q(D))) \cup \cdots \cup \mathcal{F}(\langle\, a_i \leftarrow e_i \,\rangle, \phi_{var,N})(\sigma_{\psi_N \wedge \phi_{det,N}}(Q(D)))$$

where $\phi_{var,i}$ and $\phi_{det,i}$ are respectively the non-deterministic and deterministic clauses of ϕ (i.e., $\phi = \phi_{var,i} \wedge \phi_{det,i}$) for each partition. Partitioning then, consists of two stages: (1) Obtaining a set of potential partitions Ψ from the original condition ϕ, and (2) Segmenting ϕ into a deterministic filtering predicate and a non-deterministic lineage component.

4.1.1 Partitioning the Query

Algorithm 2 takes the selection predicate ϕ in the shim query $\mathcal{F}_{\langle\, a_i \leftarrow e_i \,\rangle, \phi}$, and outputs a set of partitions $\Psi = \{\psi_i\}$. Partitions are formed from the set of all possible truth assignments to a set of candidate clauses. Candidate clauses are obtained from if statements appearing in ϕ that have deterministic conditions, and that branch between deterministic and non-deterministic cases. For example, the if statement in Example 2 branches between deterministic values for non-null attributes, and non-deterministic possible replacements.

Example 8. *The normal form $\mathcal{F}(Q(D))$ of the query in the prior example has the non-deterministic condition (ϕ):*

$$(\textbf{if } brand \texttt{ is null } \textbf{then } Var('b', \texttt{ROWID}) \textbf{ else } brand) = \text{`}Apple' \wedge (\textbf{if } cat \texttt{ is null } \textbf{then } Var('c', \texttt{ROWID}) \textbf{ else } cat) = \text{`}phone'$$

There are two candidate clauses in ϕ: brand `is null` *and cat* `is null`*. Thus, Algorithm 2 creates 4 partitions: $\psi_1 = (\neg brand \texttt{ is null} \wedge \neg cat \texttt{ is null})$, $\psi_2 = (brand \texttt{ is null} \wedge \neg cat \texttt{ is null})$, $\psi_3 = (\neg brand \texttt{ is null} \wedge cat \texttt{ is null})$, and finally $\psi_4 = (brand \texttt{ is null} \wedge cat \texttt{ is null})$.*

Algorithm 2 `naivePartition`(ϕ)

In: ϕ: A non-deterministic boolean expression
Out: Ψ: A set of partition conditions $\{\psi_i\}$

```
clauses ← ∅
Ψ ← ∅
for (if condition then α else β) ∈ subexps(φ) do
  /* Check ifs in φ for candidate partition clauses */
  if isDet(condition) ∧ (isDet(α) ≠ isDet(β)) then
    clauses ← clauses ∪ {condition}
/* Loop over the power-set of clauses */
for partition ∈ 2^clauses do
  ψ_i ← ⊤
  /* Clauses in the partition are true, others are false */
  for clause ∈ clauses do
    if clause ∈ partition then ψ_i ← ψ_i ∧ clause
                          else ψ_i ← ψ_i ∧ ¬clause
  Ψ ← Ψ ∪ {ψ_i}
```

4.1.2 Segmenting ϕ

For each partition ψ_i we can simplify ϕ into a reduced form ϕ_i. We use $\phi[\psi_i]$ to denote the result of propagating the implications of ψ_i on ϕ. For example, $(\textbf{if } X \texttt{ is null } \textbf{then } Var('X') \textbf{ else } X)[X \texttt{ is null}] \equiv Var('X')$. Using `isDet` from Algorithm 1, we partition the conjunctive terms of $\phi[\psi_i]$ into deterministic and non-deterministic components $\phi_{i,det}$ and $\phi_{i,var}$, respectively so that

$$(\phi_{i,det} \wedge \phi_{i,var}) \equiv \phi[\psi_i]$$

4.1.3 Partitioning Complex Boolean Formulas

As discussed in Section 3.5 there are three cases where non-determinism can be data-dependent: conditional expressions, conjunctions, and disjunctions. Algorithm 2 naively targets only conditionals. Conjunctions come for free, because deterministic clauses can be freely migrated into the deterministic query already. However, queries including disjunctions can be further simplified.

Example 9. *We return once again to our running example, but this time with a disjunction in the* `WHERE` *clause*

```
SELECT name FROM SaneProduct
WHERE brand = 'Apple' OR cat = 'phone'
```

Propagating ψ_3 into the normalized condition ϕ gives:

$$(\phi[\psi_3]) \equiv \left(brand =' Apple' \vee Var('c', \texttt{ROWID}) =' phone'\right)$$

The output is always deterministic for rows where $brand =' Apple'$. However, this formula can not be subdivided into deterministic and non-deterministic components as above.

We next describe a more aggressive partitioning strategy that uses the structure of ϕ to create partitions where each partition depends on exactly the same set of Var terms. To

determine the set of partitions for each sub-query, we use a recursive traversal through the structure of ϕ, as shown in in Algorithm 3. In contrast to the naive partitioning scheme, this algorithm explicitly identifies two partitions where ϕ is deterministically true and deterministically false. This additional information helps to exclude cases where one clause of an OR (resp., AND) is deterministically true (resp., false) from the non-deterministic partitions. To illustrate, consider the disjunction case handled by Algorithm 3. In addition to the partition where both children are non-deterministic, the algorithm explicitly distinguishes two partitions where one child is non-deterministic and the other is deterministically false. When ϕ is segmented, the resulting non-deterministic condition for this partition will be simpler.

Algorithm 3 `generalPartition`(ϕ)

In: ϕ: A non-deterministic boolean expression.
Out: $\psi_\top$: The partition where ϕ is deterministically true.
Out: $\psi_\bot$: The partition where ϕ is deterministically false.
Out: Ψ_{var}: The *set* of non-deterministic partitions.

if ϕ **is** $\phi_1 \vee \phi_2$ **then**
  $\langle\ \psi_{\top,1}, \psi_{\bot,1}, \Psi_{var,1}\ \rangle \leftarrow$ `generalPartition`(ϕ_1)
  $\langle\ \psi_{\top,2}, \psi_{\bot,2}, \Psi_{var,2}\ \rangle \leftarrow$ `generalPartition`(ϕ_2)
  $\psi_\top \leftarrow \psi_{\top,1} \vee \psi_{\top,2}$
  $\psi_\bot \leftarrow \psi_{\bot,1} \wedge \psi_{\bot,2}$
  for all $\psi_{var,1}, \psi_{var,2} \in \Psi_{var,1}, \Psi_{var,2}$ **do**
    $\Psi \leftarrow \Psi \cup \{\psi_{var,1} \wedge \psi_{var,2}\}$
  for all $\psi_{var,1} \in \Psi_{var,1}$ **do** $\Psi \leftarrow \Psi \cup \{\psi_{var,1} \wedge \psi_{\bot,2}\}$
  for all $\psi_{var,2} \in \Psi_{var,2}$ **do** $\Psi \leftarrow \Psi \cup \{\psi_{var,2} \wedge \psi_{\bot,1}\}$
else if ϕ **is** $\phi_1 \wedge \phi_2$ **then**
  `/* Symmetric with disjunction */`
else if ϕ **is** $\neg\phi_1$ **then**
  $\langle\ \psi_{\top,1}, \psi_{\bot,1}, \Psi_{var,1}\ \rangle \leftarrow$ `generalPartition`(ϕ_1)
  $\langle\ \psi_\bot, \psi_\top, \Psi_{var}\ \rangle = \langle\ \psi_{\top,1}, \psi_{\bot,1}, \Psi_{var,1}\ \rangle$
else
  $\Psi =$ `naivePartition`(ϕ)
  $\Psi_{det} \leftarrow \emptyset$; $\Psi_{var} \leftarrow \emptyset$
  for all $\psi \in \Psi$ **do**
    if `isDet`$(\phi[\psi])$ **then** $\Psi_{det} \leftarrow \Psi_{det} \cup \{\psi\}$
    else $\Psi_{var} \leftarrow \Psi_{var} \cup \{\psi\}$
  $\psi_\top = (\bigvee \Psi_{det}) \wedge \phi[\bigvee \Psi_{det}]$
  $\psi_\bot = (\bigvee \Psi_{det}) \wedge \neg\phi[\bigvee \Psi_{det}]$

The partition approach makes full use of the backend database engine by splitting the query into deterministic and non-deterministic fragments. The lineage of the condition for each sub-query is simpler, and generally not data-dependent for all rows in a partition. As a consequence, explanation objects can be shared across all rows in the partition. The number of partitions obtained with both partitioning schemes is exponential in the number of candidate clauses. Partitions could conceivably be combined, increasing the number of redundant tuples processed by Mimir to create a lower-complexity query. In the extreme, we might have only two partitions: one deterministic and one non-deterministic. We leave the design of such a partition optimizer to future work.

4.2 Approach 2: Inline

During best-guess query evaluation, each variable instance is replaced by a single, deterministic best-guess. Simply put, best-guess queries are themselves deterministic. The second approach exploits this observation to directly offload virtually all computation into the database. Best-guess values for all variable instances are pre-materialized into the database, and the Var terms themselves are replaced by nested lookup queries that can be evaluated directly.

4.2.1 Best-Guess Materialization

As part of lens creation, best-guess estimates must now be materialized. Recall from the grammar in Figure 2, $Var(id, e_1, \ldots, e_n)$ terms are defined by a unique identifier id and zero or more parameters (e_i). For each unique variable identifier allocated by the lens, Mimir creates a new table in the database. The schema of the best-guess table consists of the variable's parameters (e_i), a best guess value for the variable, and other metadata for the variable including whether the user "Accept"ed it. The variable parameters form a key for the best-guess table.

Example 10. *Recall the domain repair lens from Example 2. To materialize the best-guess relation, Mimir run the lens query to determine all variable instances that are used in the current database instance. In the example, there are 4 such variables, one for each null value. For instance, the missing brand of product P123 will instantiate a variable $Var('brand', 'P123')$. For all "brand" variables Mimir will create a best guesses table with primary key* `param1`*, a best-guess value* `value`*, and attributes storing the additional metadata mentioned above. Mentions of variables in queries over the lens are replaced with a subquery that returns the best guess value. For example, in $\pi_{brand}(SaneProduct)$ the expression for* `brand` *in the VG-RA query:*

if *brand* `is null` **then** $Var('brand', \texttt{ROWID})$ **else** *brand*

is translated into the SQL expression

```
CASE WHEN brand IS NULL
     THEN (SELECT value FROM best_guess_brand b
           WHERE b.param1 = Product.ROWID)
     ELSE brand END
```

To populate the best guess tables, Mimir simulates execution of the lens query, and identifies every variable instance that is used when constructing the lens' output. The result of calling `getBestGuess` on the corresponding model is inserted into the best-guess table. When a non-deterministic query is run, all references to Var terms are replaced by nested lookup queries which read the values for Var terms from the corresponding best guess tables. As a further optimization, the in-lined lens query can also be pre-computed as a materialized view.

4.2.2 Recovering Provenance

This approach allows deterministic relational databases to directly evaluate best-guess queries over C-Tables, eliminating the need for a shim query $\mathcal{F}$ to produce results. However, the shim query also provides a form of provenance, linking individual results to the Var terms that might affect them. Mimir's annotated cursors rely on this link to efficiently determine whether a result row or cell is uncertain and also when constructing explanation objects.

For inlining to be compatible with annotated cursors, three further changes are required: (1) To retain the ability to quickly determine whether a given result row or column is deterministic, result relations are extended with a 'determinism' attribute for the row and for each column. (2)

To quickly construct explanation objects, we inject a provenance marker into each result relation that can be used with the shim query $\mathcal{F}$ to quickly reconstruct any row or cell's full provenance. (3) To count the number of potentially missing rows, we initiate a secondary arity-estimation query that is evaluated off of the critical path.

4.2.3 Result Determinism

Recall from Example 6 that expressions involving conditionals, conjunctions, and disjunctions can create situations where the determinism of a row or column is data dependent. In the naive execution strategy, these situations arise exclusively in the shim query $\mathcal{F}$ and can be easily detected. As the first step towards recovering annotated cursors, we push this computation down into the query itself.

Concretely, we rewrite a query Q with schema $sch(Q) = \{a_i\}$ into a new query $[[Q]]_{det}$ with schema $\{a_i, D_i, \phi\}$. Each D_i is a boolean-valued attribute that is true for rows where the corresponding a_i is deterministic. ϕ is a boolean-valued attribute that is true for rows deterministically in the result set. We refer to these two added sets of columns as attribute- and row-determinism metadata, respectively. Query $[[Q]]_{det}$ is derived from the input query Q by applying the operator specific rewrite rules described below, in a top-down fashion starting from the root operator of query Q.

Projection. The projection rewrite relies on a variant of Algorithm 1, which rewrites columns according to the determinism of the input. Consequently, the only change is that the column rewrite on line 5 replaces columns with a reference to the column's attribute determinism metadata:

4: **else if** E **is** $Column_i$ **then**
5: **return** D_i

The rewritten projection is computed by extending the projection's output with determinism metadata. Attribute determinism metadata is computed using the expression returned by `isDet` and row determinism metadata is passed-through unchanged from the input.

$$[[\pi_{a_i \leftarrow e_i}(Q)]]_{det} \mapsto \pi_{a_i \leftarrow e_i, D_i \leftarrow \texttt{isDet}(e_i), \phi \leftarrow \phi}([[Q]]_{det})$$

Selection. Like projection, the selection rewrite makes use of `isDet`. The selection is extended with a projection operator that updates the row determinism metadata if necessary.

$$[[\sigma_{\psi}(Q)]]_{det} \mapsto \pi_{a_i \leftarrow a_i, D_i \leftarrow D_i, \phi \leftarrow \phi \wedge \texttt{isDet}(\psi)}(\sigma_{\psi}([[Q]]_{det}))$$

Cross Product. Result rows in a cross product are deterministic if and only if both of their input rows are deterministic. Cross products are wrapped in a projection operator that combines the row determinism metadata of both inputs, while leaving the remaining attributes and attribute determinism metadata intact.

$$[[Q_1 \times Q_2]]_{det} \mapsto \pi_{a_i \leftarrow a_i, D_i \leftarrow D_i, \phi \leftarrow \phi_1 \wedge \phi_2}([[Q_1]]_{det} \times [[Q_2]]_{det})$$

Union. Bag union already preserves the determinism metadata correctly and does not need to be rewritten.

$$[[Q_1 \cup Q_2]]_{det} \mapsto [[Q_1]]_{det} \cup [[Q_2]]_{det}$$

Relations. The base case of the rewrite, once we arrive at a deterministic relation, we annotate each attribute and row as being deterministic.

$$[[R]]_{det} \mapsto \pi_{a_i \leftarrow a_i, D_i \leftarrow \top, \phi \leftarrow \top}(R)$$

Optimizations. These rewrites are quite conservative in materializing the full set of determinism metadata attributes at every stage of the query. It is not necessary to materialize every D_i and ϕ if they can be computed statically based solely on each operator's output. For example, consider a given D_i that is data-independent, as in a deterministic relation or an attribute defined by a Var term. D_i has the same value for every row, and can be factored out of the query. A similar property holds for Joins and Selections, allowing the projection enclosing the rewritten operator to be avoided.

4.2.4 Explanations

Recall that explanation objects provide a way to analyze the non-determinism in a given result row or cell. Given a query $Q(D)$ and its normalized form $\mathcal{F}(Q'(D))$, this analysis requires only $\mathcal{F}$ and the individual row in the output of $Q'(D)$ used to compute the row or cell being explained.

We now show how to construct a provenance marker during evaluation of Q and how to use this provenance marker to reconstruct the single corresponding row of Q'. The key insight driving this process is that the normalization rewrites for cross product and union (Rewrites 3 and 4 in Figure 3) are isomorphic with respect to the data dependency structure of the query; Q and Q' both have unions and cross products in the same places.

As the basis for provenance markers, we use an implicit, unique per-row identifier attribute called `ROWID` supported by many popular database engines. When joining two relations in the in-lined query, their `ROWID`s are concatenated (we denote string concatenation as $\circ$):

$$Q_1 \times Q_2 \mapsto \pi_{a_i \leftarrow a_i, \texttt{ROWID} \leftarrow \texttt{'('} \circ \texttt{ROWID}_1 \circ \texttt{')('} \circ \texttt{ROWID}_2 \circ \texttt{')'}}(Q_1 \times Q_2)$$

When computing a bag union, each source relation's `ROWID` is tagged with a marker that indicates which side of the union it came from:

$$Q_1 \cup Q_2 \mapsto \pi_{a_i \leftarrow a_i, \texttt{ROWID} \leftarrow \texttt{ROWID} \circ \texttt{'+1'}}(Q_1) \cup \pi_{a_j \leftarrow a_j, \texttt{ROWID} \leftarrow \texttt{ROWID} \circ \texttt{'+2'}}(Q_2)$$

Selections are left unchanged, and projections are rewritten to pass the `ROWID` attribute through.

The method `unwrap`, summarized in Algorithm 4, illustrates how a symmetric descent through the deterministic component of a normal form query and a provenance marker can be used to produce a single-row of Q'. The descent unwraps the provenance marker, recovering the single row from each join leaf used to compute the corresponding row of Q'.

4.3 Approach 3: Hybrid

The first two approaches provide orthogonal benefits. The partitioning approach results in faster execution of queries over deterministic fragments of the data, as it is easier for the backend database query optimizer to take advantage of indexes already built over the raw data. The inlining approach results in faster execution of queries over non-deterministic fragments of the data, as joins over non-deterministic values do not create a polynomial explosion of possible results. Our third and final approach is a simple combination of the two: Queries are first partitioned as in Approach 1, and then non-deterministic partitions are in-lined as in Approach 2.

5. EXPERIMENTS

We now summarize our the results of experimental analysis of the two optimizations presented in this paper. We

Algorithm 4 $\texttt{unwrap}(Q', id)$

In: Q': The deterministic component of a VG-RA normal form query.
In: id: A `ROWID` from the inlined query Q that was normalized into $\mathcal{F}(Q'(D))$.
Out: A query to compute row id of Q'

```
if Q' is π(Q1) then
  return π(unwrap(Q1, id))
else if Q' is σ(Q1) then
  return σ(unwrap(Q1, id))
else if Q' is Q1 × Q2 and id is (id1)(id2) then
  return unwrap(Q1, id1) × unwrap(Q2, id2)
else if Q' is Q1 ∪ Q2 and id is id1+1 then
  return unwrap(Q1, id1)
else if Q' is Q1 ∪ Q2 and id is id1+2 then
  return unwrap(Q2, id1)
else if Q' is R then
  return σ_{ROWID=id}(R)
```

evaluate Virtual C-Tables under the classical normalization-based execution model, and partition-, inline-, and hybrid-optimized execution models. All experiments are conducted using both SQLite as a backend, and a major commercial database termed DBX due to its licensing agreement. Mimir is implemented in Scala and Java. Measurements presented are for Mimir's textual front-end. All experiments were run under RedHat Enterprise Linux 6.5 on a 16 core 2.6 GHz Intel Xeon server with 32 GB of RAM and a 4-disk 900 GB RAID5 array. Mimir and all database backends were hosted on the same machine to avoid including network latencies in measurements. Our experiments demonstrate that: (1) Virtual C-Tables scale well, (2) Virtual C- Tables impose minimal overhead compared to deterministic evaluation, and (3) Hybrid evaluation is typically optimal.

5.1 Experimental Setup

Datasets were constructed using TPC-H [9]'s dbgen with scaling factors 1 (1 GB) and 0.1 (100 MB). To simulate incomplete data that could affect join predicates, we randomly replaced a percentage of foreign key references in the dataset with `NULL` values. We created domain constraint repair lenses over the damaged relations to "repair" these `NULL` values as non-materialized views. As a query workload, we used TPC-H Queries 1, 3, 5, and 9 modified in two ways. First, all relations used by the query were replaced by references to the corresponding domain constraint repair lens. Second, Mimir does not yet include support for aggregation. Instead we measured the cost of enumerating the set of results to be aggregated by stripping out all aggregate functions and computing their parameter instead[1]. Execution times were capped at 30 minutes.

We experimented with two different backend databases: **SQLite** and a major commercial database **DBX**. We tried four different evaluation strategies: **Classic** is the naive, normalization-based evaluation strategy, while **Partition**, **Inline**, and **Hybrid** denote the optimized approaches presented in Sections 4.1, 4.2, and 4.3 respectively. **Deterministic** denotes the four test queries run directly on the backend databases with un-damaged data, and serves as an *lower bound* for how fast each query can be run.

[1]The altered queries can be found at https://github.com/UBOdin/mimir/tree/master/test/tpch_queries/noagg

5.2 Comparison

Figures 6 and 7 show the performance of Mimir running over SQLite and DBX, respectively. The graphs show Mimir's overhead relative to the equivalent deterministic query.

Table scans are unaffected by Mimir. Query 1 is a single-table scan. In all configurations, Mimir's overhead is virtually nonexistent.

Partitioning accelerates deterministic results. Query 3 is a 3-way foreign-key lookup join. Under naive partitioning, completely deterministic partitions are evaluated almost immediately. Even with partitioning, non-deterministic subqueries still need to be partly evaluated as cross products, and partitioning times out on all remaining queries.

Partitioning can be harmful. Query 5 is a 6-way foreign-key lookup join where Inline performs better than Hybrid. Each foreign-key is dereferenced in exactly one condition in the query, allowing Inline to create a query with a plan that can be efficiently evaluated using Hash-joins. The additional partitions created by Hybrid create a more complex query that is more expensive to evaluate.

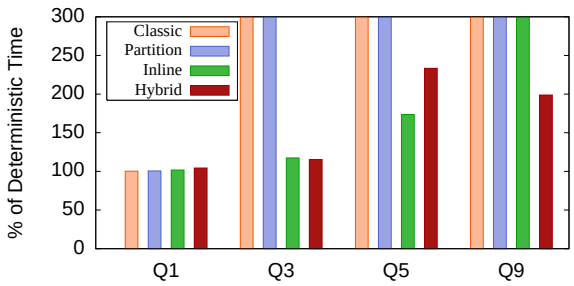
Partitioning can be helpful. Query 9 is a 6-way join with a cycle in its join graph. Both `PARTSUPP` and `LINEITEM` have foreign key references that must be joined together. Consequently, Inlining creates messy join conditions that neither backend database evaluates efficiently. Partitioning results in substantially simpler nested queries that both databases accept far more gracefully.

6. RELATED WORK

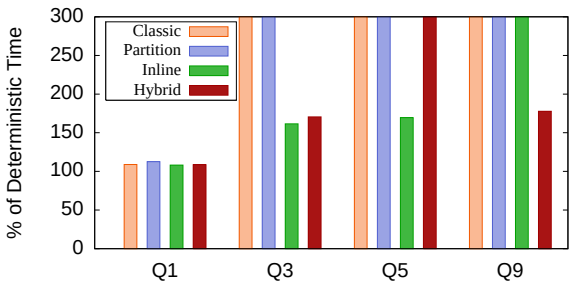
The design of Mimir draws on a rich body of literature, spanning the areas probabilistic databases, model databases, provenance, and data cleaning. We now relate key contributions in these areas on which we have based our efforts.

Incomplete Data. Enabling queries over incomplete and probabilistic data has been an area of active research for quite some time. Early research in the area includes `NULL` values [8], the C-Tables data model [17] for representing incomplete information, and research on fuzzy databases [37]. The C-Tables representation used by Mimir has been linked to both probability distributions and provenance through so-called PC-Tables [15] and Provenance Semirings [21], respectively. These early concepts were implemented through a plethora of probabilistic database systems. Most notably, MayBMS [16] employs a simplification of C-Tables called U-Relations that does not rely on labeled nulls, and can be directly mapped to a deterministic relational database. However, U-Relations can only encode uncertainty described by finite discrete distributions (e.g., Bernoulli), while VC-Tables can support continuous, infinite distributions (e.g., Gaussian). Other probabilistic database systems include MCDB [18], Sprout [13], Trio [1], Orion 2.0 [31], Mystiq [6], Pip [22], Jigsaw [23], and numerous others. These systems all require heavyweight changes to the underlying database engine, or an entirely new database kernel. By contrast, Mimir is an external component that attaches to an existing deployed database, and can be trivially integrated into existing deterministic queries and workflows.

Model Databases. A specialized form of probabilistic databases focus on representing structured models such as graphical models or markov processes as relational data. These types of databases exploit the structure of their mod-

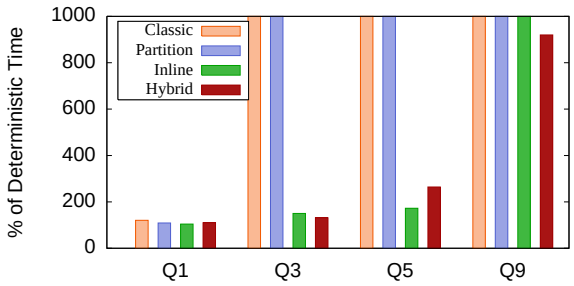


(a) TPC-H SF 0.1 (100 MB)

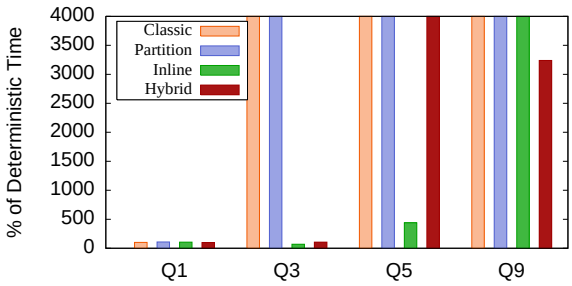


(b) TPC-H SF 1 (1 GB)

Figure 6: Performance of Mimir running over SQLite as a percent of deterministic query execution time.



(a) TPC-H SF 0.1 (100 MB)



(b) TPC-H SF 1 (1 GB)

Figure 7: Performance of Mimir running over DBX as a percent of deterministic query execution time.

els to accelerate query evaluation. Systems in this space include BayesStore [33], MauveDB [10] Lahar [25], and SimSQL [7]. In addition to defining semantics for querying models, work in this space typically explores techniques for training models on deterministic data already in the database. The Mimir system treats lens models as black boxes, ignoring model structure. It is likely possible to incorporate model database techniques into Mimir. We leave such considerations as future work.

Provenance. Provenance (sometimes referred to as lineage) describes how the outputs of a computation are derived from relevant inputs. Provenance tools provide users with a way of quickly visualizing and understanding how a result was obtained, most commonly as a way to validate outliers, better understand the results, or to diagnose errors. Examples of provenance systems include a general provenance-aware database called Trio [1], a collaborative data exchange system called Orchestra [14], and a generic database provenance middleware called GProM which also supports updates and transactions [2, 3]. It has been shown that certain types of provenance can encode C-Tables [21]. It is this connection that allows Mimir to provide reliable feedback about sources of uncertainty in the results. The VG-Relational algebra used in Mimir creates symbolic expressions that act as a form of provenance similar to semiring provenance [21] and its extensions to value expressions (aggregation [21]) and updates [3].

Data Curation. In principle, it is useful to query uncertain or incomplete data directly. However, due to the relative complexity of declaring uncertainty upfront, it is still typical for analysts to validate, standardize, and merge data *before* importing it into a data management system for analysis. Common problems in the space of data curation include entity de-duplication [11, 28, 30], interpolation [10, 26], schema matching [4, 27, 24, 29], and data fusion. Mimir's lenses each implement a standard off-the-shelf curation heuristic. These heuristics usually require manual tuning, validation, and refinement. By contrast, in Mimir these difficult, error-prone steps can be deferred until query-time, allowing analysts to focus on the specific cleaning tasks that are directly relevant to the query results at hand.

Other systems for simplifying or deferring curation exist. For example, DataWrangler [20] creates a data cleaning environment that uses visualization and predictive inference to streamline the data curation process. Similar techniques could be used in Mimir for lens creation, streamlining data curation even further. Mimir can also trace its roots to on-demand cleaning systems like Paygo [19], CrowdER [34], and GDR [35]. In contrast to Mimir, these systems each focus on a specific type of data cleaning: Duplication and Schema Matching, Deduplication, and Conditional Functional Dependency Repair, respectively. Mimir provides a general curation framework that can incorporate the specialized techniques used in each of these systems.

Uncertainty Visualization. Visualization of uncertain or incomplete data arises in several domains. As already noted, DataWrangler [20] uses visualization to help guide users to data errors. MCDB [18] uses histograms as a summary of uncertainty in query results. Uncertainty visualization has also been studied in the context of Information Fusion [32, 5]. Mimir's explanation objects are primitive by comparison, and could be extended with any of these techniques.

7. CONCLUSIONS

We presented Mimir, an on-demand data curation system based on a novel type of probabilistic data curation operators called Lenses. The system sits as a shim layer on-top of a relational DBMS backend - currently we support SQLite and a commercial system. Lenses encode the result of a curation operation such as domain repair or schema match-

ing as a probabilistic relation. The driving force behind Mimir's implementation of Lenses are VC-Tables which are a representation of uncertain data that cleanly separates the existence of uncertainty from a probabilistic model for the uncertainty. This enables efficient implementation of queries over lenses by outsourcing the deterministic component of a query to the DBMS. Furthermore, the symbolic expressions used by VC-Tables to represent uncertain values and conditions act as a type of provenance that can be used to explain how uncertainty effects a query result. In this paper we have introduced several optimizations of this approach that 1) push part of the probabilistic computation into the database (we call this *inlining*) without loosing the ability to generate explanations and 2) partitioning a query based on splitting selection conditions such that some fragments can be evaluated deterministically or can benefit from available indexes. In future work we will investigate cost-based optimization techniques for lens queries and using the probabilistic model for uncertainty in the database to exclude rows that are deterministically not in the result (e.g., if a missing brand value is guaranteed to be either Apple or Samsung, then this value does not fulfill a condition $brand = Sony$).

8. REFERENCES

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, 2006.

[2] B Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. A generic provenance middleware for database queries, updates, and transactions. *TaPP*, 2014.

[3] Bahareh Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. Reenacting transactions to compute their provenance. Technical report, Illinois Institute of Technology, 2014.

[4] Philip A Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 2011.

[5] Ann M. Bisantz, Richard Finger, Younho Seong, and James Llinas. Human performance and data fusion based decision aids. In *FUSION*, 1999.

[6] Jihad Boulos, Nilesh N. Dalvi, Bhushan Mandhani, Shobhit Mathur, Christopher Ré, and Dan Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *SIGMOD*, 2005.

[7] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, 2013.

[8] E. F. Codd. Extending the database relational model to capture more meaning. *ACM TODS*, 4(4):397–434, 1979.

[9] Transaction Processing Performance Council. TPC-H specification. http://www.tpc.org/tpch/.

[10] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, 2006.

[11] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, January 2007.

[12] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: Getting to the core. In *PODS*, pages 90–101. ACM, 2003.

[13] Robert Fink, Andrew Hogue, Dan Olteanu, and Swaroop Rath. Sprout^2: a squared query engine for uncertain web data. In *SIGMOD*, 2011.

[14] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Provenance in ORCHESTRA. *DEBU*, 33(3):9–16, 2010.

[15] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. *IEEE Data Eng. Bull.*, 29(1):17–24, 2006.

[16] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. MayBMS: a probabilistic database management system. In *SIGMOD*, pages 1071–1074. ACM, 2009.

[17] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[18] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[19] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, pages 847–860. ACM, 2008.

[20] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI*, 2011.

[21] G. Karvounarakis and T.J. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[22] Oliver Kennedy and Christoph Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.

[23] Oliver Kennedy and Suman Nath. Jigsaw: efficient optimization over uncertain enterprise data. In *SIGMOD*, 2011.

[24] Yoonkyong Lee, Mayssam Sayyadian, AnHai Doan, and Arnon S Rosenthal. etuner: tuning schema matching software using synthetic scenarios. *VLDB J.*, 16(1):97–122, 2007.

[25] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *ICDE*, March 2009.

[26] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. Eracer: A database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[27] Robert McCann, Warren Shen, and AnHai Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, 2008.

[28] Fabian Panse, Maurice van Keulen, and Norbert Ritter. Indeterministic handling of uncertain decisions in deduplication. *JDIQ*, 4(2):9:1–9:25, March 2013.

[29] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[30] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

[31] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne Hambrusch, and Rahul Shah. Orion 2.0: Native support for uncertain data. In *SIGMOD*, pages 1239–1242. ACM, 2008.

[32] Ion-George Todoran, Laurent Lecornu, Ali Khenchaf, and Jean-Marc Le Caillec. A methodology to evaluate important dimensions of information quality in systems. *J. DIQ*, 6(2-3):11:1–11:23, June 2015.

[33] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.

[34] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[35] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

[36] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. Lenses: An on-demand approach to etl. *PVLDB*, 8(12):1578–1589, 2015.

[37] Maria Zemankova and Abraham Kandel. Implementing imprecision in information systems. *Information Sciences*, 37(1):107 – 141, 1985.