Lenses: An On-Demand Approach to ETL

Ying Yang†, Niccolò Meneghetti†, Ronny Fehling°, Zhen Hua Liu†, Oliver Kennedy†
† SUNY Buffalo, ° Oracle
{yyang25, niccolom, okennedy}@buffalo.edu
{ronny.fehling, zhen.liu}@oracle.com

ABSTRACT

Three mentalities have emerged in analytics. One view holds that reliable analytics is impossible without high-quality data, and relies on heavy-duty ETL processes and upfront data curation to provide it. The second view takes a more ad-hoc approach, collecting data into a data lake, and placing responsibility for data quality on the analyst querying it. A third, on-demand approach has emerged over the past decade in the form of numerous systems like Paygo or HLog, which allow for incremental curation of the data and help analysts to make principled trade-offs between data quality and effort. Though quite useful in isolation, these systems target only specific quality problems (e.g., Paygo targets only schema matching and entity resolution). In this paper, we explore the design of a general, extensible infrastructure for on-demand curation that is based on probabilistic query processing. We illustrate its generality through examples and show how such an infrastructure can be used to gracefully make existing ETL workflows “on-demand”. Finally, we present a user interface for On-Demand ETL and address ensuing challenges, including that of efficiently ranking potential data curation tasks. Our experimental results show that On-Demand ETL is feasible and that our greedy ranking strategy for curation tasks, called CPI, is effective.

1. INTRODUCTION

Effective analytics depends on analysts having access to accurate, reliable, high-quality information. One school of thought on data quality manifests as Extract-Transform-Load (ETL) processes that attempt to shield analysts from any uncertainty, by cleaning all data thoroughly up-front. The cleansed data is usually represented in a new or transformed way as tables in a data warehouse. Those tables, typically in form of star schemas, as well as the transformation and parsing logic, all have to be designed up front, an arduous and time-consuming task. Only after loading the parsed and transformed data into the data warehouse can the analyst query the data to do any actual analysis. We collectively refer to this selective parsing, transformation and loading into a new structure as data curation.

Example 1. Alice is an analyst at the HappyBuy retail store, and is developing a promotional strategy based on public opinion ratings for its products gathered by two data collection companies. A thorough analysis of the data requires substantial data-cleaning effort from Alice: As shown in Figure 1, the rating companies schemas are incompatible, and HappyBuy’s own product data is incomplete. However, Alice’s preliminary analysis is purely exploratory, and she is hesitant to invest the full effort required to curate this data.

The upfront costs of curation have lead many to instead inline curation tasks into the analytical process, so that only immediately relevant curation tasks are performed.

Example 2. Alice realizes that she only needs two specific attributes for her analysis: category and rating. She considers manually constructing a task-specific data set containing a sanitized version of only these two columns.

This deferred approach is more lightweight, but encourages analysts to develop brittle one-off data cleansing solutions, incurring significant duplication of effort or organizational overheads. A third approach, initially explored as part of Paygo [25], instead curates data incrementally in response to specific query requirements. This form of on-demand curation results in a sanitized data set that is based on a principled trade-off between the quality desired from the data set and the human effort invested in curating it. Paygo specifically targets two curation tasks: schema matching and entity resolution, and other systems have since appeared for schema matching [2], as well as other tasks like information extraction [10], and inference [41,42].

A typical ETL pipeline often involves many distinct curation tasks, requiring that multiple on-demand data curation systems be used in tandem. However, the data representations and quality metrics used by these systems are optimized for very specific use-cases, making composition difficult. In this paper, we explore and address the challenges of composing specialized on-demand curation techniques into a general-purpose workflow. The result is a unified model for on-demand curation called On-Demand ETL that bridges the gap between these systems and allows them to be gracefully incorporated into existing ETL and analytics workflows. This unified model builds around ordinary SQL, retaining compatibility with existing standards for ETL design, data analysis, and database management.
Representing Incomplete Data. On-demand curation permits trade-offs between data quality, and the effort needed to obtain high-quality data. This requires a representation for the quality loss incurred by only partially curating data. Existing on-demand curation systems use specialized, task-specific representations. In Section 2 we describe an existing representation for incomplete information called PC-Tables [17, 18, 23], and show how it can be leveraged by On-Demand ETL.

Expressing Composition. If the output of a curation technique is non-deterministic, then for closure, it must accept non-deterministic input as well. In Section 3, we define a model for non-deterministic operators called lenses that capture the semantics of on-demand data curation processes. We illustrate the generality of this model through examples, and show that it is closed over PC-Tables.

Backwards Compatibility. For On-Demand ETL to be practical, it must be compatible with traditional data management systems and ETL pipelines. In Section 4, we develop a practical implementation of PC-Tables [23] called Virtual C-Tables that can be safely embedded into a classical, deterministic database system or ETL workflow.

Presenting Data Quality. In Section 5, we discuss how to present the quality loss incurred by incomplete curation to end-users. We show how lightweight summaries can be used to alert an analyst to specific problems that affect their analysis, and how On-Demand ETL computes a variety of quality measures for query results.

Feedback. Section 6 highlights how lenses act as a form of provenance, linking uncertainty in query outputs to the lenses that created them. These links allow for lens-defined curation tasks that improve the quality of query results. We introduce a concept called the cost of perfect information (CPI) that relates the value of a curation task that improves a result’s quality, to the cost of performing the task, allowing curation tasks to be ranked according to their net value to the analyst.

Experimental Results. Finally, in Section 7, we present experimental results that demonstrate the feasibility of On-Demand ETL and provide several insights about its use.

Concretely, this paper’s contributions include: (1) A composable model for expressing data curation tasks based on probabilistic components called lenses, (2) A practical implementation of PC-Tables called Virtual C-Tables that can be deployed into classical databases without needing support for labeled nulls, (3) A family of heuristics called CPI used to prioritize curation tasks that improve result quality at a cost, and (4) Experimental results that illustrate the feasibility of On-Demand ETL and the effectiveness of CPI.

2. BACKGROUND AND RELATED WORK

A deterministic database is a finite collection of relation instances \( \{R_1, \ldots, R_k\} \) over a schema \( S = \{S_1, \ldots, S_k\} \). According to the “possible worlds” semantics [37] a probabilistic database \( D \) consists of a pair \((W, P)\), where \( W \) is a large collection of deterministic databases, the so called possible worlds, all sharing the same schema \( S \), and \( P \) is a probability measure over \( W \). Roughly speaking, \( D \) is a database whose schema is known but whose internal state is uncertain, and \( W \) simply enumerates all its plausible states. We denote by \( R \) the set of all tuples that appear in some possible world (often called possible tuples). Each element of \( R \) is an outcome for the probability space \((W, P)\). The confidence of a possible tuple \( t \) is simply the probability that it will appear in the database \( D \), i.e. its marginal probability

\[
P(t \in D) = \sum_{W_i \subseteq W} P(W_i)
\]

The goal of probabilistic databases \([1, 8, 16, 22, 24, 27, 34, 36]\) is to support the execution of deterministic queries like regular, deterministic databases do. Let’s denote by \( Q \) an arbitrary deterministic query (i.e., a query expressible in classical bag-relational algebra) and by \( sch(Q) \) the schema defined by it, which consists of a single relation. The application of \( Q \) to \( D \), denoted by \( Q(D) \), generates a new probability space \((W', P')\) where \( W' = \{Q(W_i) \mid W_i \in W\} \) and

\[
P'(t \in Q(D)) = \sum_{W_i \in W : \exists i \in \text{supp}(Q)} P(W_i)
\]

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, or by computing expectations for aggregate values. These tasks are difficult in practice, mainly for two reasons: (i) \( W \) is usually too large to be enumerated explicitly, and (ii) computing marginals is provably #P-hard in the general case. For example, if our schema contains a single relation and our set of possible worlds contains all subsets of a given set of 100 tuples, then we have \( 2^{100} \) distinct possible worlds where each possible tuple appears in half.

One way to make probabilistic query processing efficient is to encode \( W \) and \( P \) with a compact, factorized representation. In this paper we adopt a generalized form of C-Tables [23, 27] to represent \( W \), and PC-Tables [17, 18] to represent the pair \((W, P)\). A C-Table [23] is a relation instance where each tuple is annotated with a lineage formula \( \phi \), a propositional formula over an alphabet of variable symbols \( \Sigma \). The formula \( \phi \) is often called a local condition and
the symbols in $\Sigma$ are referred to as labeled nulls, or just variables. Intuitively, for each assignment to the variables in $\Sigma$ we obtain a possible relation containing all the tuples whose formula $\phi$ is satisfied. For example:

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>brand</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>P125</td>
<td>Apple 6s, White</td>
<td>Apple</td>
<td>phone</td>
</tr>
<tr>
<td>P125</td>
<td>Samsung Note2</td>
<td>Samsung</td>
<td>phone</td>
</tr>
</tbody>
</table>

The above C-Table defines a set of three possible worlds, $\{t_1, t_2, t_3\}$, $\{t_3\}$, and $\{t_2\}$, 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. C-Tables are closed w.r.t. positive relational algebra [23]: if $W$ is representable by a C-Table and $Q$ is a positive query then $W^Q = \{Q(W_i) \mid W_i \in W\}$ is representable by another C-Table.

Following the approach of PIP [27], in this paper we adopt VG-RA (variable-generating relational algebra), a generalization of positive bag-relation algebra with extended projection, that uses a simplified form of VG-functions [24]. In VG-RA, VG-functions (i) dynamically introduce new Skolem symbols in $\Sigma$, 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 C-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: \textit{Var}(\ldots).

From PIP, we also inherit a slightly generalized form of C-Tables. Our C-Tables differ from the canonical ones in the following: (i) Variables in $\Sigma$ are allowed to range over continuous domains, (ii) attribute-level uncertainty is encoded by replacing missing values with VG-RA expressions (not just functions) that act as Skolem terms and (iii) these VG-RA expressions allow basic arithmetic operations. The previous example is equivalent to the PIP-style C-Table:

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>brand</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>P125</td>
<td>Apple 6s, White</td>
<td>Var(\textit{A} $\neq$ 1)</td>
<td>phone</td>
</tr>
<tr>
<td>P125</td>
<td>Samsung Note2</td>
<td>Samsung</td>
<td>phone</td>
</tr>
</tbody>
</table>

From now on, without explicitly mentioning PIP, we will assume all C-Tables support the generalizations discussed above. It has been shown that C-Tables are closed w.r.t VG-RA [23, 27]. The semantics for VG-RA query evaluation $[\textit{C}]_CT$ over C-Tables [22, 23, 27] are summarized in Figure 3. These semantic rules make extensive use of the lazy evaluation operator $[\textit{C}]_\textit{lazy}$, which uses a partial binding of \textit{Column} or \textit{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 $\phi$ is preserved intact through the projection. Selection is evaluated by combining the selection predicate $\phi$ with each tuple’s existing local condition. As an optimization, tuples for which $\phi$ deterministically evaluates to false (⊥) are preemptively discarded.

A PC-Table [17, 18] is a C-Table augmented with a probability measure $P$ over the possible assignments to the variables in $\Sigma$. Since each assignment to the variables in $\Sigma$ generates a possible world, a PC-Table induces a probability measure over $W$. Hence, it can be used to encode a probabilistic database $(W, P)$. PC-Tables are the foundation for several PQP systems, including MayBMS [22], Orchestra [16] and PIP [27]. Green et al. [17] observed that PC-Tables generalize other models like Trio [1]. The relationship between PC-Tables and VG-RA is discussed in further detail in Section 3.

### 2.1 Other Related Work

There are numerous tools for on-demand data curation, each targeting specific challenges like schema matching [2, 25], de-duplication [25], information extraction [9, 10], information integration [5], or ontology construction [4, 19]. On-Demand ETL generalizes these, providing a tool for on-demand data curation that these solutions can be plugged into. On-demand curation can also be thought of as a highly-targeted form of crowd-sourced databases [32], which leverage the power of humans to perform complex tasks.

The problem of incomplete data arises frequently in distributed systems, where node failures are common. Existing solutions based on uncertainty [7, 15, 30, 40] can similarly be expressed in On-Demand ETL to provide more fine-grained analyses of result quality over partial data than these approaches provide natively.

Prioritizing curation tasks, as addressed in Section 6, is quite closely related to Stochastic Boolean Function Evaluation, or SBFE, where the goal is to determine the value of a given Boolean formula by paying a price to discover the exact value of uncertain boolean variables. The problem is hard in its general form; exact solutions and heuristics have been proposed for several classes of functions [26, 39]. More recently Deshpande et al. [11] designed a 3-approximation algorithm for linear threshold formulas, while Allen et al. [3] developed exact and approximate solutions for monotone k-term DNF formulas.

### 3. LENSES

A lens is a data processing component that is evaluated as part of a normal ETL pipeline. Unlike a typical ETL processing stage that produces a single, deterministic output, a lens instead produces a PC-Table $(W, P)$, which defines the set of possible outputs, and a probability measure that approximates the likelihood that any given possible output accurately models the real world. In effect, a lens provides structure to uncertainty about how any given possible output data may be interpreted by the ETL pipeline.

Asking ETL designers to specify this structure manually for the entire ETL process is impractical. Lenses allow this structure to be specified as a composition of individual simple transformations, constraints, or target properties that take the place of normal operators in the ETL pipeline. However, composition requires closure. In this section, we define a closed framework for lens specification, and illustrate its generality through three example lenses.

### 3.1 The Lens Framework

A lens instance is defined over a query $Q(D)$, and is in turn responsible for constructing a PC-Table $(W, P)$. A lens defines $W$ as a C-Table through a VG-RA expression
The C-Table for the lens' output is constructed by the query over C-Tables, so the lens constructs $P$ as a joint probability distribution over every variable introduced by $\mathcal{F}_lens(Q(D))$ by defining a sampling process in the style of classical VG-functions [24], or supplementing it with additional metadata to create a PIP-style grey-box [27]. An example of the complete process for the Domain Constraint Lens defined below is illustrated in Figure 4. 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. VG-RA is closed over C-Tables, so $\mathcal{F}_lens(Q(D))$ simply defines a new C-Table. Defining $P$ as an extension of $P_Q$ with distributions for all variables newly introduced by $\mathcal{F}_lens$ provides closure for the probability measure, a topic we will return to in Section 3.3.

### 3.2 Lens Examples

We first illustrate the generality of the lens framework through three example lenses: domain constraint repair, schema matching, and archival. To construct a lens over query $Q$, the user writes:

```sql
CREATE LENS <lens_name> AS Q
USING <lens_type>(<lens_arguments>);
```

**Domain Constraint Repair.** A domain constraint repair lens enforces attribute-level constraints such as NOT NULL. Under the assumption that constraint violations are a consequence of data-entry errors or missing values, domain constraint violations can be repaired by finding a legitimate replacement for each invalid value. Obtaining reliable replacement values typically requires detailed domain knowledge. However, in an on-demand setting, approximations are sufficient. The domain constraint repair lens uses educated guesses about data domains (e.g., uniform distributions over allowable values) and machine learning models (e.g., a naive Bayes classifier trained over $Q(D)$) to approximate domain knowledge. With $sch(Q) = \{ \{ a_1, t_1 \}, \ldots, \{ a_n, t_n \} \}$ denoting the attributes $a_i$ of $Q(D)$ and their type $t_i$, a domain constraint repair lens definition has the form:

```sql
... USING DOMAIN_REPAIR(a_1, t_1, ..., a_n, t_n)
```

The C-Table for the lens’ output is constructed by the query $\mathcal{F}_lens = \pi_{(\ldots, a_i \leftarrow V_i, \ldots)}$, where each $V_i$ is defined as:

- $V_i$ is defined as
- $\text{if } t_i = a_i \text{ then } a_i \text{ else } Var(Name_{i,ROWID})$

In this expression, $t_i = a_i$ if $a_i$ satisfies the type constraints of $t_i$, and each $Name_{i}$ is a freshly allocated variable name. Independently, $P$ is defined by training a classifier or similar model for each attribute on the output of $Q$.

**Example 3. Returning to Example 1, Alice creates a lens to handle missing values in the Product table:**

```sql
CREATE LENS SaneProduct AS SELECT * FROM Product
USING DOMAIN_REPAIR( category 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 output of the query over `Product`. Figure 5 shows the C-Table for this lens.

**Schema Matching.** A schema matching lens creates a mapping from the source data’s schema to a user-defined target schema. This is especially important for non-relational data like JSON objects or web tables, which may not have well-defined schemas [21, 25]. Given a destination schema $\{ \{ b_1, t_1 \}, \ldots, \{ b_m, t_m \} \}$ and a threshold $\omega$, a schema matching lens definition has the form:

```sql
... USING SCHEMA_MATCHING(b_1, t_1, ..., b_m, t_m, \omega)
```

The schema matching lens defines a fresh boolean variable $Var(Name_{i,j})$ for every pair $a_i, b_j$, where $(a_i, t_i) \in sch(Q)$. The probability of $Var(Name_{i,j})$ corresponds to the probability of a match between $a_i$ and $b_j$. $\mathcal{F}_lens$ takes the form: $\pi_{(\ldots, b_j \leftarrow V_j, \ldots)}$, where $V_j$ enumerates possible matches for $b_j$:

- $\text{if } Var(Name_{i,j}) \text{ then } a_i \text{ else }$
- $\vdots$
- $\text{if } Var(Name_{n,j}) \text{ then } a_n \text{ else NULL}$

As an optimization, matches for type-incompatible pairs of attributes are skipped. Additionally, the lens discards matches where the likelihood of a schema-level match falls below a user-defined threshold ($\omega$).

**Example 4. Alice next turns to the ratings data sets, which have incompatible schemas. She creates a lens and a joint view:**

```sql
CREATE LENS MatchedRatings2 AS SELECT FROM Ratings2
USING SCHEMA_MATCHING( pid string, ..., rating float, review_ct float, NO LIMIT );
CREATE VIEW AllRatings AS SELECT FROM MatchedRatings2
UNION SELECT FROM Ratings1;
```
The resulting C-Table for MatchedRatings2 is shown in Figure 6. From Alice’s perspective, AllRatings behaves as a normal view combining Ratings1 and Ratings2. Behind the scenes, the attributes of Ratings2 are quietly matched against those of Ratings1. In this example, only evaluation and num_ratings are type compatible match candidates, and other match cases are dropped.

Numerous options are available for constructing \( P \), including domain-based schemes or complex ontology-based matching. However, even a simple matching scheme can be sufficient for On-Demand ETL. We approximate the probability of a match between two attributes by a normalized edit distance between the two attribute names. As we show in Section 7 (Figure 11), even this simple matcher can produce suitable results.

Archival. An archival lens captures the potential for errors arising from OLAP queries being posed over stale data [29], like queries run in between periodic OLTP to OLAP bulk data copies. The lens takes a list of pairs \( \langle T, R \rangle \), where \( R \) is a reference to a relation in an OLTP database, and \( T \) is the period with which \( R \) is periodically archived.

\[
... \text{USING ARCHIVAL}((T_1, R_1), \ldots, (T_m, R_m))
\]

This lens probabilistically discards rows from its output that are no-longer valid according to the lens query \( F_{\text{lens}} = \sigma_{V_{\text{var}}(\text{Name}, R_{\text{colID}})} \) where \( \text{Name} \) is a freshly allocated identifier. In the background, the lens periodically polls for samples drawn from each \( R_i \) to estimate the volatility of each relation referenced by \( Q \). Denote by \( \nu_j \) the probability of a tuple in \( R_i \) being invalidated at some point during the period \( T_j \). \( P \) is defined independently for each row as a binomial distribution with probability \( \prod_{(j,R_i \in Q)} \nu_j \).

### 3.3 Composing Lenses

**Example 5.** When Alice examines AllRatings, she suddenly realizes that the data in Ratings1 is missing rating information. She creates a domain repair lens:

\[
\text{CREATE LENS SaneRatings AS}
\]

\[
\text{SELECT pid, category, rating, review_ct FROM AllRatings USING DOMAIN_REPAIR(rating DECIMAL NOT NULL)}
\]

The C-Table for SaneRatings is straightforward to construct, as both lenses involved can be expressed as VG-RA expressions. However, the domain repair lens must still train a model to fill in distributions for missing values. In contrast to Example 3, where the model was trained on deterministic input, here the input is a PC-Table.

The closure of VG-RA over PC-Tables requires that any non-deterministic query \( F \) be defined alongside a process that extends the input PC-Table’s probability measure \( P_n \) to cover any variables introduced by \( F \). For lenses, there are three possibilities. In the trivial case where \( F \) introduces no new variables, \( P_n \) remains unmodified. In the second case, variables introduced by \( F \) are independent of \( P_n \), and a joint distribution is defined trivially as the product of the original and new distributions. If any new variables depend on \( P_n \), a grey-box distribution definition [27] can be used to express these dependencies directly.

However, it may not always be possible to explicitly define dependencies, particularly when adapting existing on-demand cleaning solutions. On-Demand ETL provides three separate mechanisms to enable support for lenses that require deterministic inputs: (i) Train the lens on the most-likely output of the source lens (see Section 5), (ii) Train the lens on samples of rows drawn from random instances of the source model, or (iii) Train the lens on the subset of the source data that is fully deterministic (i.e., certain).

**Example 6.** Alice issues the following query:

\[
\text{SELECT p.pid, p.category, r.rating, r.review_ct FROM SaneRatings r NATURAL JOIN Product p WHERE p.category IN (‘phone’, ‘TV’) OR r.rating > 4}
\]

The resulting C-Table is shown in Figure 7. The first two products are entirely deterministic. P125 is a phone and deterministically satisfies the query, but has attribute-level uncertainty from schema matching (Example 4). P34234 has a missing category (Example 3) and rating (Example 5), so the row’s condition is effectively the entire selection predicate. P34234 and P34235 are laptops and fail the test on category, so their presence in the result set depends entirely on how rating is matched (Example 4). Recall that there are three candidates: evaluation, num_ratings, or neither. In the last case, domain repair (Example 5) replaces the NULL with \( \text{Var}(‘Z’, R11) \) and \( \text{Var}(‘Z’, R12) \). P34234 and P34235 have functional if expressions in their conditions, with the form \( (\text{if } \phi_1 \text{ then } e_2 \text{ else } e_3 ) > 4 \). These expressions can be simplified by recursively pushing the comparison into the branches: \( \text{if } \phi_1 \text{ then } e_2 > 4 \text{ else } e_3 > 4 \), in-lining the branches into the condition: \( \phi_1 \land (e_2 > 4) \lor \neg \phi_1 \land (e_3 > 4) \), and then further simplifying the resulting boolean expression.

### 4. VIRTUAL C-TABLES

We next address the challenge of deploying the PQP techniques necessary to support Lenses into an existing database or ETL pipeline. Our approach, called Virtual C-Tables or VC-Tables, works by decomposing VG-RA queries into deterministic and non-deterministic components. Non-deterministic components are percolated out of queries, making it possible for the bulk of the ETL process to remain within a classical deterministic system. A small On-Demand ETL shim layer wraps around the database, and provides a minimally-invasive interface for uncertainty-aware users and applications. This shim layer is also responsible for managing several views, discussed in Section 5, that provide backwards compatibility for legacy applications.

Let \( \mathcal{F}(D) \) denote a VG-RA query over a deterministic database \( D \). When combined with a probability measure \( P \), \( \langle \mathcal{F}(D), P \rangle \) defines a PC-Table. Semantics for deterministic queries over PC-Tables are well defined, but rely on support for labeled nulls, a feature not commonly found in popular data management systems. Existing probabilistic query processing systems address this limitation by restricting themselves to special cases like finite-discrete probability distributions over categorical data [13, 22], relying on costly user-defined types [24, 27, 28], or by specializing the entire database for uncertain data management [1, 16, 36]. Ultimately, each of these solutions is either too specialized for On-Demand ETL, or too disruptive to be deployed into an existing classical ETL pipeline or databases.

Virtual C-Tables decouple the deterministic components of a query from the non-deterministic components that define a PC-Table. This decomposition is enabled by the observation that once the probability measure \( P \) of a PC-Table
\[
\pi_{a' \leftarrow a}(F((a \leftarrow e), \phi)(Q(D))) \\
\equiv F((a' \leftarrow [if \ src = 1 \ then \ e1 \ else \ e1]'_a)_{azy}, \phi), (Q(D))) \quad (1)
\]

\[
\pi_{a' \leftarrow a}, \phi(F((a \leftarrow e), \phi)(Q(D))) \\
\equiv F((a \leftarrow e), \phi \land \psi_{var} \land \psi_{desc})(Q(D))) \quad (2)
\]

\[
F((a \leftarrow e), \phi)(Q(D)) \cup F((a \leftarrow e'), \phi')(Q(D)) \\
\equiv F((a \leftarrow e, a' \leftarrow e'), \phi \land \phi')(Q(D) \times Q'(D)) \quad (3)
\]

\[
F((a \leftarrow e), 1, \phi(Q(D))) \psi \pi_{\star,src=2}(Q(D)) \\
\equiv F((a \leftarrow e), [if \ src = 1 \ then \ e1 \ else \ e1]'_a, [if \ src = 1 \ then \ \phi \ then \ \phi']_{azy}) \quad (4)
\]
Table 1: Example C-Table

<table>
<thead>
<tr>
<th>id</th>
<th>category</th>
<th>rating</th>
<th>review_ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>P123</td>
<td>phone</td>
<td>4.5</td>
<td>50</td>
</tr>
<tr>
<td>P124</td>
<td>phone</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>P125</td>
<td>phone</td>
<td>2*</td>
<td>3*</td>
</tr>
<tr>
<td>P4225</td>
<td>laptop</td>
<td>5*</td>
<td>4.5*</td>
</tr>
</tbody>
</table>

(Up to 2 results may be missing. *)

Figure 9: The best-guess summary of the C-Table from Figure 7 that Alice actually sees.

5.1 Summarizing the Result Relation

Users consume a Virtual C-Table \( F(\langle a_i \leftarrow e_i \rangle, \phi)(Q(D)) \) through one of two deterministic summary relations: A deterministic relation \( R_{det} \), and a best-guess relation \( R_{guess} \). The deterministic relation \( R_{det} \) represents the certain answers \([12]\) of the virtual C-Table, and is constructed by replacing every variable reference in each \( e_i \) and \( \phi \) with NULL, and dropping rows where \( \phi \neq T \):

\[
\text{SELECT } e_i(* \rightarrow \text{NULL}) \text{ AS } a_i \text{ FROM } Q(D) \text{ WHERE } \phi(* \rightarrow \text{NULL})
\]

The resulting relation contains all of the rows deterministically present in \( F(Q(D)) \), with NULL taking the place of any non-deterministic values. \( R_{det} \) can be computed entirely within a classical deterministic database, making it backwards compatible with legacy ETL components.

The best-guess relation \( R_{guess} \) is constructed in two stages. First, the deterministic database system executes \( Q(D) \). As the classical database streams results for \( Q(D) \), the shim layer evaluates each \( e_i \) and \( \phi \) based on the valuation given by \( \text{argmax}(P(v)) \), the most-likely possible world. Field values or row confidences in the best guess relation that depend on \( v \) are annotated in the shim layer’s output. Legacy applications can quietly ignore this annotation. In uncertainty-aware applications, this annotation is used to indicate which parts of the result are uncertain to the end-user.

Example 7. Continuing Example 6, the database now responds to Alice’s query with the most-likely result shown in Figure 9. Every non-deterministic (i.e., guessed) field is annotated with an asterisk. Every row with a non-deterministic condition is similarly marked. A footer indicates how many rows were dropped due to a non-deterministic condition evaluating to false in the most likely possible world. Note that this best-guess estimate is not entirely accurate: evaluation has been mapped to review_ct, and rating has not been matched, resulting in best-effort guesses of 2 and 5 for the last two rows of the result. In spite of the error, Alice can quickly see the role uncertainty plays in her results.

5.2 Summarizing Result Quality

Once the user is made aware that parts of a query result may be uncertain, two questions likely to be asked are “How bad?” and “Why?”. Answering the latter question is trivial: \( F \) contains a reference to all of the variables that introduce uncertainty, each of which is explicitly linked to the lens that constructed it. In other words, \( F \) serves as a form of provenance that can be used to explain sources of uncertainty to the end-user.

The former question requires us to develop a notion of result quality. Our approach is based on the ideas of noise: intuitively, the less noise is present in the model, the higher the quality of the best-guess relation’s predictions. We abstractly define result quality as the level of confidence that the user should have in the annotated best-guess results.

These results include both non-deterministic attribute values, as well as possible tuples.

Recall that a non-deterministic value in \( R_{guess} \) is obtained from non-deterministic expressions in \( R_{guess} \). Numerous metrics that effectively convey the quality of attribute values drawn from a probabilistic database have been proposed, including pessimistic hard bounds \([29]\), variance \([24,27]\), and \( \epsilon - \delta \) bounds \([20]\).

As a simplification, we assume that with respect to understanding uncertainty in a specific attribute value, the cognitive burden on the user is constant, while for the presence or absence of rows in the output, it scales linearly. Intuitively, guessing wrong about tuple presence can mean the difference between overwhelming the user with a flood of unnecessary results, and hiding the presence of potentially critical information. Under this assumption, tuple-level uncertainty adds more noise to the result, and we will focus primarily on this type of uncertainty from here on.

The appearance of a tuple in the query result is determined by the ground truth of its local condition \( t.\phi \). Valuations \( v(\Sigma) \) map \( t.\phi \) to a deterministic boolean value \( t.\phi(v) \). From the PC-Table’s probability measure \( P(v) \), we get the binomial distribution \( P(t.\phi(v)) \), often called the confidence of \( t \). We use the confidence of \( t \) to measure how difficult it is for the analyst to predict the ground truth of \( t.\phi \). Intuitively, if \( P(t.\phi(v)) \) is skewed towards 0 or 1, we expect to predict the value of \( t.\phi \) with reasonable accuracy; on the other hand, if \( P(t.\phi(v)) \) is a fair coin flip, we have no reliable information about the expected result of \( t.\phi \). It is natural to use Shannon entropy as a metric to quantify the quality of the query result. We define the entropy of a tuple in terms of its confidence \( p_t = P(t.\phi(v)) \) as:

\[
\text{entropy}(t) = -(p_t \cdot \log_2(p_t) + (1 - p_t) \cdot \log_2(1 - p_t))
\]

Efficiently approximating tuple confidences by sampling from \( P(v) \) is well studied in probabilistic databases \([22, 34]\), and we use similar techniques for estimating tuple entropies.

To unify the individual per-attribute and per-row metrics, we define a relation-wise noise function \( N(R) \) as a linear combination of individual metrics. For example, a relation \( R \) without non-deterministic attributes might have \( N(R) = \sum_{t \in R} \text{entropy}(t) \). To account for the entropy generated by non-deterministic attributes, we start with the intuition that each attribute in the output provides \( \frac{1}{N} \)th of the information content of a tuple, where \( N \) is the arity of \( R \). Thus, by default, we assume each non-deterministic value contributes to the noise seen in the final result a fraction in the range \([0, \frac{1}{N}] \) inversely proportional to the attribute’s estimated variance.

6. FEEDBACK

When the analyst is given a query result \( R \) that does not meet her quality expectations, she can allocate additional resources for gathering more evidence. For example, she may spend some time gathering ground-truth values for variables in the output C-Table. By construction, variables represent uncertainty about basic facts. For example, a schema matching lens generates expressions of the form \( \text{Var}(\text{rat} = \text{eval}) \) that could be stated as a simple question like “Do rating and evaluation mean the same thing?”. Replacing variables with their ground truths means performing these basic curation tasks, with the goal of reducing the
6.1 Prioritizing Curation Tasks

Prioritizing curation tasks is a dynamic decision process, as the outcome of one curation task affects the choice of the next one to be performed. Let’s assume, for the moment, that the analyst has no budget constraints and her goal is simply to determine the ground truth of a given condition formula $\phi$, minimizing the expected amount of resources spent in the process. In the literature, this optimization problem is known as stochastic boolean function evaluation [11, 39]. Both exact and approximated algorithms have been proposed for several classes of formulas. In general, the problem can be thought of as a Markov Decision Process\(^2\), having one state for each partial assignment to the variables in $\phi$ and one action for each variable (a curation task). Rewards are determined by $-c(\cdot)$ and state-transitions are determined by $P(v)$. Final states consist of assignments that make $\phi$ either true or false with certainty. The planning horizon is finite and equal to the number of variables in $\phi$. A simple solution to the problem consists of a policy, prescribing a curation task for each non-terminating assignment to perform. The application of a policy is an interactive process: the system instructs the analyst to address a particular curation task (“Do rating and evaluation mean the same thing?”), the analyst provides the required ground truth, and asks the system for the next move. This feedback loop continues until the deterministic value of $\phi$ is obtained. As a baseline for evaluation (Section 7), On-Demand ETL implements a naive algorithm for computing policies of this kind, named naive minimum expected total cost (NMETC).

6.2 Balancing Result Quality and Cost

Real-world ETL applications are unlikely to be free from budget constraints. Even when budget is not a problem, the average analyst will rarely aim for perfect information. Instead, she would rather target a reasonable approximation of the value of $\phi$, setting an upper-bound on the entropy of the formula. Hence, we generalize the approach discussed above and make the assumption that the analyst wants to plan her curation tasks so to maximize a hidden value function $V(\cdot)$, which depends on $c(\cdot)$ and $N(\cdot)$ and is unknown to the system. Clearly, we assume $V(\cdot)$ decreases monotonically as the cumulative cost increases, and increases monotonically as the noise decreases. In simple words, $V$ determines how much the analyst is willing to pay for an improvement in the estimation of the value of $\phi$, on a case-by-case basis. We call this trade-off the cost of perfect information (CPI). Since the details of $V(\cdot)$ are unknown, the goal of the system is to propose several candidate policies. Each policy should guarantee a certain expected entropy at the price of a certain expected cumulative cost. The user is then able to choose the candidate policy that best matches her hidden value function. Since the analyst may be subject to budget constraints hidden to the system, the candidate list includes greedy versions of the policies, computed progressively over limited planning horizons. Inspired by the algorithms EG2 [33], CS_ID3 [38] and CS,C4.5 [14], On-Demand ETL supports the following four greedy strategies:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$\text{CPI}(v_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EG2</td>
<td>$\left(2^{I(G(R[v_i])} - 1\right)/c_i$</td>
</tr>
<tr>
<td>CS_ID3</td>
<td>$I(G[R(v_i)])/\sqrt{c_i}$</td>
</tr>
<tr>
<td>CS_IDX</td>
<td>$I(G[R(v_i)]/c_i$</td>
</tr>
<tr>
<td>CS,C4.5</td>
<td>$I(G[R(v_i)])/c_i$</td>
</tr>
</tbody>
</table>

Here, $IG$ denotes the information gain, or the reduction in noise produced by the curation task on variable $v_i$.

Example 8. Consider the condition $\phi$ for P34235 in Figure 7, which has the form $A \lor (B \land C)$. Figure 10 illustrates the decision tree that ranks curation tasks (the three variables), given lens-defined ground-truth costs and marginal probabilities as shown in the figure. The expected entropy after performing the curation task for $v_i$, denoted by $E[H(v_i)]$, is computed as a weighted average over all possible outcomes of the task. $\text{CPI}(v_i)$ is computed according to the CS_ID3 formula given above, with $IG[R(v_i)] = H - E[H(v_i)]$.

7. EXPERIMENTS

In this section we show the feasibility of On-Demand ETL and explore several points in its design space. Specifically,
we find that: (i) The greedy approach of minimizing CPI produces higher-quality query results at lower costs than optimizing for total cost when the hidden value function is not known, (ii) The precise formula used to compute CPI is not critical to achieving high quality results, (iii) When composing lenses, order is relevant, as open-ended lenses like domain-constraint repair can fix issues created by other lenses earlier in the pipeline, and (iv) Tree-based classifiers work best for domain constraint repair lenses.

7.1 Experimental Setup

Our experimental setup consists of three data sets drawn from publicly available sources. To simulate data-entry error, a portion of the data values are randomly removed. To simulate an analyst’s querying behavior, we identify one attribute in each data set, remove the attribute from the source data, and use a tree-based classifier to construct a query that the analyst might issue to recover the attribute. For each data source, we also provide simulated user-defined costs for available curation tasks.

Product Data. We used the product search APIs of two major electronics retailers to extract product data for a total of 586 items (346 and 240 items respectively). The products extracted fall into three categories: TVs, cell phones and laptops. There are ten attributes in the schema of each data source. We randomly replaced 45% of the data values with NULL, and coerce both data sets into the schema of a third retailer’s search API. On this data-set, we simulate an analyst trying to predict what factors go into a good product rating. We trained a tree based classifier on the partial data, used the resulting decision tree to simulate the analyst’s query:

```sql
SELECT * FROM products
WHERE brand in (4,5,6,7) AND category in (1,2,3)
AND totalReviews < 3 AND instoreAvailability = 0
AND (onsale Clearance = 0 OR (quantityAvailableHint = 0
AND shippingCost in (0,1,2,3,4)));
```

Curation tasks fall into four categories: Trivial schema matching tasks, simple data gathering of boolean values like item availability, more detailed data gathering of values like strings, and more open-ended data gathering tasks such as soliciting item reviews from focus groups. We assign the cost of these four curation tasks to be 1, 5, 10, and 30 units of effort respectively.

Credit Data. We used the German and Japanese Credit Data-sets from the UCI data repository. These data sets contain 1000 and 125 items, respectively, and have 10 and 8 attributes, respectively. As in the product dataset, we randomly replaced 45% of data values with NULL values. The German data is coerced into the schema of the Japanese data set. We simulate an analyst searching for low-risk customers by using the following classifier-constructed query:

```sql
SELECT * FROM PD
WHERE (purchase_item < 0.5 AND monthly_payment >= 3.5
AND num_of_years_in_company in (2,3)
OR (num_of_months >= 6.5 AND married_gender >= 2.5));
```

In addition to trivial schema-matching tasks, there are two kinds of missing attributes: Some attributes can be computed from other values (e.g., a customer’s monthly payment can be computed from the total loan value and duration) or retrieved from other parts of the bank. Other attributes require personal information about the client. We set the cost of these three classes of task to be 1, 10, and 20 respectively.

Real Estate Data. We obtain house listing information from five real estate websites. Unlike the prior cases, where the number of data-sets is small and the number of records per data set is comparatively large, the Real Estate data set emulates web-tables where the number of data sets is comparatively large and the number of records per data set is small. We further reduce the data size by randomly sampling only 20 items from each dataset. As above, 45% of data values are replaced by NULL. All source data is coerced into a globally selected target-schema. For this data set, we simulate an analyst trying to identify houses likely to have a price rating of 3 out of 4 points, where all curation tasks have a flat cost of 1.

```sql
SELECT * FROM PD WHERE Baths < 2.5
AND (Beds >= 3.5 OR Garage >= 2.5);
```

7.2 Lens Configuration

For each experiment, we simulate an analyst using the Domain-Constraint Repair and Schema Matching lenses described in Section 3.2. We use the values randomly removed from each data set as ground truths for evaluating the Domain-Constraint Repair lens, and a manually defined mapping as ground truth for evaluating the Schema Matching lens. Both lenses define curation tasks as summarized in the description of each data set.

Schema Matching. We employ a combination of schema matchers that hybridize structure-level and element-level matchers. We first use constraint-based (data type and data range) filters to eliminate candidate matches, and then use the Levenshtein, JaroWinkler, and Ngram distance metrics to rank attribute matches based on string similarity with a threshold. The performance of these three strategies is shown in Figure 11. We take an average of the similarity scores from the three distance metrics and normalize them to approximate the probability of a match.

Domain Constraint Repair. We use incremental classifiers from the massive online analysis (MOA) framework for the Domain-Constraint Repair lens. We use classifiers in five categories: active, bayes, stochastic gradient descent, ensemble and tree. For each attribute in the source table, we train a classifier using tuples in which the value is not missing. The estimation results for missing values are probability distributions of all candidate repairs.

7.3 Ranking Curation Tasks

We compare three ranking policies over curation tasks (one per variable vi). Each policy implements a ranking over the available curation tasks, the top-ranked task is performed. Curation costs are as listed in Section 7.1.

NMETC. This (naive, exponential-time) policy calculates an optimal long-term strategy based on repeatedly selecting the variable that minimizes the global expected total cost of obtaining a deterministic value. Potential curation tasks are ranked in descending order of their expected total cost, weighted over all possible paths through the decision tree.

---

3http://developer.bestbuy.com/documentation/products-api
4http://developer.walmartlabs.com/docs/read/Search_API
5http://go.developer.ebay.com
6http://pages.cs.wisc.edu/~anhai/wisc-si-archive/domains/real_estate1.html
Figure 12: Composability of schema matching and domain repair for 11 classifiers (Product Data)

Figure 13: Composability of schema matching and domain repair for 11 classifiers (Credit Data)

Figure 14: Composability of schema matching and domain repair for 11 classifiers (Real Estate Data)

Figure 15: Performance comparison for different methods on query results of (a) product, (b) credit, and (c) real estate data-sets. Detailed step-by-step performance for the naive strategy is computationally infeasible for the real estate data-set, so only final results are shown.
Greedy (CPI). CPI based policies rank curation tasks in ascending order of CPI. All four CPI-based metrics produce virtually identical results for each of our test cases, so only results for the EG2 implementation of CPI are shown. The scoring function for the greedy strategy is the CPI itself.

Random. The random strategy ranks curation tasks in a random order, and provides a baseline for other methods.

7.4 Lens Composition

We first explore the default (i.e., pre-feedback) behavior of lenses under composition. Of interest to us are three questions: (i) Does the machine learning model used for domain-constraint repair matter? (ii) Can lenses be composed together safely? and (iii) Does the order in which lenses are composed matter? We evaluate the accuracy of the output of Schema Matching (SM) and Domain Constraint Repair (DCR) lenses applied to each data set. Figures 12, 13, and 14 show the fraction of cells in the output of each query that correspond to ground truth results before any feedback is gathered. Our results include two variants of Domain-Constraint Repair, one where all data sources are combined before being repaired (DCR-Joint), and one where all data sources are repaired independently (DCR-Sep). We consider three different lens combinations: DCR-Joint or DCR-Sep applied to the output of SM (DCR-Joint ← SM and DCR-Sep ← SM, respectively), and SM applied to the output of DCR-Sep (SM ← DCR-Sep). The remaining combination is not possible, as DCR-Joint requires SM first to create a unified schema. For comparison, we also present results for each lens alone, using ground truth values for the output of the other lens. Performance results are shown for 11 different machine learning models from the MOA framework.

In general, the performance of different orderings of lenses appears to differ by only a small amount, generally under 5%. An exception appears in the Product data set (Figure 12), where we can see for all estimation methods, applying SM first and then applying DCR-Joint produces the best results. By being trained on both data-sets together, DCR is able to detect and correct some schema matching errors. Moreover, in all cases, the combined error of composing both lenses is lower than the error introduced by either lens individually. This shows that composing different lenses is feasible. By comparison, the Credit data set (Figure 13) is extremely noisy — both lenses have initial error rates around 34%. Hence, too much noise exists in the data, and different lens orderings have little effect.

The observation above shows that reordering lenses can be beneficial in some cases. Given analyses of lenses, we can help users reorder lenses to achieve better accuracy. Another observation is that when the data is sufficiently correlated for DCR to have relatively small error rates, the error rate of DCR-Joint is typically lower than DCR-Sep. Intuitively, if inter-attribute correlations from different data sets are similar, DCR-Joint is effectively being trained on a larger dataset.

7.5 On-Demand ETL

We next study the effectiveness of On-Demand ETL and CPI-based heuristics. To study the efficacy of our CPI-based approach, we investigate the performance of different ranking strategies on product, credit, and real estate data-sets. We use the same basic setup as described above for each data-set. We present results using DCR-Joint applied to SM, but all three composition orders for DCR-Sep produce high-quality results with minimal investment.

NMETC denotes the naive brute-force cost-optimization strategy, while Random denotes a completely random ordering of curation tasks. Although the brute-force strategy produces a completely reliable result at the lowest cost, it does so at the expense of short-term benefits. For the product data-sets, a result with virtually no entropy is reached after 24,000 units of cost, while the brute force strategy requires over 30,000 units. Although NMETC requires the lowest cost to obtain a deterministic query result, it may not be optimal for a limited budget or when the user’s value function is not known.

8. CONCLUSIONS

We have presented On-Demand ETL, which generalizes task-specific on-demand curation solutions such as Paygo. On-Demand ETL enables composable non-deterministic data processing operators called Lenses that provide the illusion of fully cleaned relational data that can be queried using standard SQL. Lenses use PC-Tables to encode output, and can be deployed in traditional, deterministic database environments using Virtual C-Tables. On-Demand ETL supports best-effort guesses at the contents a PC-Table, evaluation of quality measures over a PC-Table, and a family of heuristics for prioritizing curation tasks called CPI. We have demonstrated the feasibility and need for On-Demand ETL, and the effectiveness of CPI-based heuristics.

9. REFERENCES


