Reptile: Aggregation-level Explanations for Hierarchical Data

Zezhou Huang
Columbia University
New York, NY, USA
zh2408@columbia.edu

Eugene Wu
Columbia University
New York, NY, USA
ewu@cs.columbia.edu

ABSTRACT

Users often can see from overview-level statistics that some results look "off", but are rarely able to characterize even the type of error. Reptile is an iterative human-in-the-loop explanation and cleaning system for errors in hierarchical data. Users specify an anomalous distributive aggregation result (a *complaint*), and Reptile recommends drill-down operations to help the user "zoom-in" on the underlying errors. Unlike prior explanation systems that intervene on raw records, Reptile intervenes by learning a group's expected *statistics*, and ranks drill-down sub-groups by how much the intervention fixes the complaint. This group-level formulation supports a wide range of error types (missing, duplicates, value errors) and uniquely leverages the distributive properties of the user complaint. Further, the learning-based intervention lets users provide domain expertise that Reptile learns from.

In each drill-down iteration, Reptile must train a large number of predictive models. We thus extend factorised learning from count-join queries to aggregation-join queries, and develop a suite of optimizations that leverage the data's hierarchical structure. These optimizations reduce runtimes by >6× compared to a Lapack-based implementation. When applied to real-world Covid-19 and African farmer survey data, Reptile correctly identifies 21/30 (vs 2 using existing explanation approaches) and 20/22 errors. Reptile has been deployed in Ethiopia and Zambia, and used to clean nation-wide farmer survey data; the clean data has been used to design na tional drought insurance policies.

ACM Reference Format:
Zezhou Huang and Eugene Wu. 2022. Reptile: Aggregation-level Explanations for Hierarchical Data. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3514221.3517854

1 INTRODUCTION

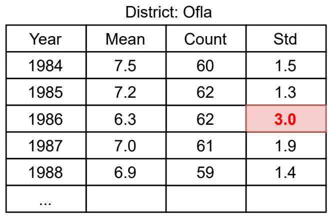
Data exploration tools follow the "overview, zoom, then details" analysis pattern [53] to help users analyze their datasets at a high level before diving into the individual records. However, modern datasets are often hierarchical and multi-dimensional. Thus, when users identify an anomalous aggregate value that is too high or too low (complaints in an overview), it can be difficult to know which attributes to drill-down (zoom in) and which of the drill-down results to focus on. This is particularly relevant when anomalous results are due to systematic data errors (e.g., missing or duplicate records, measurement errors) that the user wants to find and address. To help user drill-down, query explanation systems recommend predicates over the input database (*explanations*) that, if intervened upon, would most resolve the complaint.

Previous query explanation systems are designed for specific types of interventions, such as deletion interventions [15, 59, 72] or cell-value updates [42, 49]. But as we show in our evaluations (Section 5.2), these approaches are not applicable to errors that the interventions are not designed for. These types of bottom-up approaches seek to directly intervene on the input database records (the *bottom*), and then assess their effects on the complaint (the *up*). Unfortunately, real-world errors are unpredictable and varied (missing or duplicate records, systematic biases), and often require custom interventions. In these settings, the user must both identify the erroneous subpopulation and infer the appropriate intervention. This underspecified problem requires a human-in-the-loop approach that guides the user towards the data errors and empowers the user to provide a range of external evidence based on their domain expertise. To illustrate, we start with an example based on our collaborators at Columbia University's Financial Instruments Sector Team (FIST):

Example 1. *FIST surveys and collects drought severity data from farmers in villages throughout African countries (e.g., Ethiopia) to develop sustainable drought insurance plans for the countries. As a toy example, Figure 1a shows drought severity (from 1 (not severe) to 10 (severe)) statistics between 1984 to 1988 collected from the Ofla District in Ethiopia. The FIST researcher complains that the standard deviation in 1986 is higher than expected and suspects bias in the reporting. There are too many raw records to read manually, so the researcher wants to incrementally drill-down and narrow the scope.*

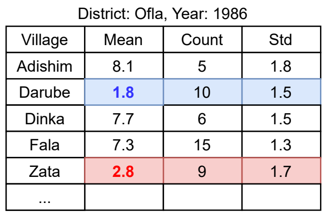
How should the researcher drill-down? Although most villages in Ofla reported high severity, Darube and Zata have low means that may contribute to the complaint (Figure 1b). The FIST researcher expects Darube's low severity because of the high rainfall in the auxiliary sensing data (Figure 1c). In contrast, Zata's low serverity is uncorroborated and thus surprising.

This example illustrates unique challenges for the FIST researcher, who wants to find a subgroup (e.g., village) along some dimension to focus attention on, and the limitations of bottom-up approaches, which attempt to repair raw data in a given village (Figure 1d) and rank repairs by their effects on the complaint (how much Ofla's Std in 1986 will reduce). These approaches fail for multiple reasons. First, record-level error detection [19, 35, 43, 58] is fundamentally ambiguous. For instance, Zata in 1986 contains many low drought severity records, however those caused by reporting


District: Ofla

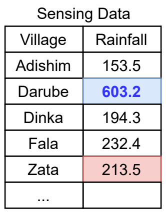
Year	Mean	Count	Std
1984	7.5	60	1.5
1985	7.2	62	1.3
1986	6.3	62	3.0
1987	7.0	61	1.9
1988	6.9	59	1.4
...			

(a) Drought statistics per year in District Ofla.


District: Ofla, Year: 1986

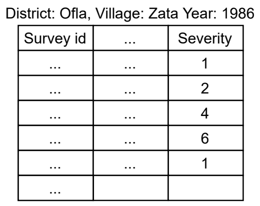
Village	Mean	Count	Std
Adishim	8.1	5	1.8
Darube	1.8	10	1.5
Dinka	7.7	6	1.5
Fala	7.3	15	1.3
Zata	2.8	9	1.7
...			

(b) After drill-down by geography to village statistics.


Sensing Data

Village	Rainfall
Adishim	153.5
Darube	603.2
Dinka	194.3
Fala	232.4
Zata	213.5
...	

(c) Auxiliary sensing dataset.


District: Ofla, Village: Zata Year: 1986

Survey id	...	Severity
...	...	1
...	...	2
...	...	4
...	...	6
...	...	1
...		

(d) Raw data of drought severity for Zata in 1986.

Figure 1: Example FIST use case. (a) The researcher thinks that Ofla's 1986 standard deviation of severity is suspiciously high (the complaint). (b) After drilling down to villages, Darube and Zata have low means and are potential explanations. (c) Darube is explained away by the high rainfall in the sensing data. (d) Raw data of drought severity for Zata in 1986 that causes the complaint.

bias (errors) are indistinguishable from those due to normal variation (clean). Second, the FIST researcher doesn't know the error type up-front, so cannot pick from the plethora of potential repair methods [19, 20, 46, 57, 71]. The real error was that farmers confused flood with drought in that year (data drift), however if the researcher incorrectly assumed the error was due to outliers [48], she might only repair raw data with the lowest severity, which was biased towards Darube. Finally, bottom-up approaches cannot detect population errors such as under-sampling [21, 67] since the records are all correct.

Existing top-down explanation and cleaning techniques are also ill-suited to this problem. Provenance summarization methods [6, 9, 36, 60] define "interestingness" measures that summarize query provenance but may be irrelevant to the complaint. For instance, Smart Drill Down's coverage measure is a special case of complaints—it translates to a complaint that a COUNT aggregation result is too high, which is not the same as the researcher's complaint. Most query explanation systems use sensitivity-based methods [59, 72] and support complaints over general aggregation functions. However, they are limited to deletion interventions, which are not always desirable—deleting Darube's data most reduces the `Std` but is inappropriate. Other explanation systems like Macrobase [5, 11] are based on risk-ratio measures that distinguish user-provided inliers and outliers. For instance, FIST may specify the 1986 result as the outlier and other results as inliers, however Macrobase would simply report rainfall shared by the inliers but not outliers, which does not explain the high variance in drought severity. Finally, counterbalancing [51] looks for sibling aggregates that offset (counterbalance) the deviation specified in the user's complaint. However, a village's drought is typically not offset by higher rainfall elsewhere.

`Reptile` tackles the above limitations. Given a hierarchical dataset and a complaint over a distributive aggregation result[1] (e.g., the std of Olfa is too high), `Reptile` recommends the next drill-down dimension and highlights the top sub-groups that affected the complaint. `Reptile` operates in a top-down fashion by learning and applying interventions to *sub-group statistics* (e.g., repair mean, count and std of Zata) rather than individual records. The distributive nature of the complaint enables `Reptile` to efficiently compute the intervention's effects on the complaint (e.g. how much `Std` in Ofla will be reduced). With this top-down approach, FIST researcher can iteratively make complaints, drill-down, and assess sub-groups. She can also verify or provide domain expertise at every step. Unlike bottom-up approaches, this top-down approach is more declarative, in that it suggests how a group's statistics *should* change, and users don't have to make low-level decisions (error detection and repair) at the raw data level. These group-level statistics also naturally express a wide range of error types (missing, duplicates, value errors) and give the FIST researcher flexibility in how they wish to intervene on the raw data. Further, this approach leverages natural hierarchies present in real-world data.

To predict appropriate aggregation-level intervention, `Reptile` relies on domain expertise from users in the form of auxiliary data (such as the sensing data) to fit a machine learning model. These types of auxiliary data have already been commonly utilized to predict statistics by data scientists across fields such as social and policital science [26], public health [68], and education [27]. For instance, public health uses county-level restaurant, population, and income statistics to estimate obesity [50]; sociology uses country-level GDP, motor vehicle density, temperature, and literacy rates to estimate physical activity [13]; and economics uses country-level infrastructure, manufacturing, and regulatory statistics to to estimate innovation [40]. Further, data augmentation tools [16] are increasingly used specifically to search and identify promising auxiliary data. We note that auxiliary datasets differ from joined tables in traditional query explanation. Joined tables are used to expand the set of attribute used to compose query explanations [59], while auxiliary datasets are used to infer the appropriate intervention.

`Reptile` makes two primary contributions based on the major challenges. The first is the scarcity of training examples when ranking the immediate groups in a candidate drill-down operation. For example, Olfa only contains a handful of villages, and its 1986 data are insufficient to train an accurate model. `Reptile` expands the training set by using parallel groups (e.g., in other years, districts, regions). However, variation across the parallel groups can severely degrade its accuracy. For instance, the villages in each district may follow a linear trend, but the slope and intercept vary from district to district, and fitting a single model across districts will be inaccurate. To address this, `Reptile` uses multi-level models [23, 28]

[1]Such as mean, count, standard deviation.

that account for systematic variation between groups. Multi-level models are well-established in the social sciences for modeling hierarchical data, and our experiments also show that they are more accurate than global approaches such as linear regression models.

The second challenge is scalable model training. `Reptile` needs to train a model for every candidate drill-down, and the exponential number of drill-down paths are too large to precompute. Further, the number of training records increases exponentially as the user drill-down into the dataset. To address this, `Reptile` avoids fully materializing the feature matrix that is exponential in the number of hierarchies, and constructs a succinct factorised matrix [54] that is orders of magnitude smaller. Prior factorised learning techniques [64, 65] assume that the predicted (Y) variable is an attribute in the join inputs to exploit the redundancy across the features *and* predicted variables. In contrast, the predicted variables in `Reptile` are aggregation results and do not exhibit these redundancies. We thus extend factorised learning to this setting, and also develop a suite of novel work-sharing and caching optimizations.

As of 2021, `Reptile` has been deployed by the FIST team in Zambia and Ethiopia, and used by local and government partners to clean data collected from villages throughout the countries, and the clean data has been used to design national drought insurance policies. In a smaller-scale user study, FIST researchers submitted 22 complaints over their drought data, and `Reptile` correctly identified data errors for 20; one was ambiguous while `Reptile` partially explained the other. In addition:

- We show that using factorised learning improves end-to-end performance by 6× as compared to a traditional Lapack-based implementation, and that our set of reptile-specific optimizations reduces runtimes by 4× as compared to the prior factorised learning system LMFAO [64].
- Across missing, duplicate, and systematic value errors, we show that `Reptile` can accurately identify the top-3 groups with 70%-100% accuracy, as compared to baseline explanation and cleaning approaches that have <60% accuracy. Further, auxiliary data improves accuracy by >20% .
- Using the John Hopkins Covid-19 data, `Reptile` correctly identified 70% of the Github-reported errors, as compared to 3 – 6.6% from popular explanation (DIFF [5], and Scorpion [59, 72]) and cleaning (HoloClean [57] and Raha [46]) techniques.

Scope: `Reptile` focuses on existential errors (i.e. missing or duplicated records), and cell value errors localized to aggregated attributes. For instance, from abnormally low `SUM`, `Reptile` can detect missing records, data drift, disguised missing values (e.g. zero), etc. However, `Reptile` assumes dimensional (categorical) attributes to be correct and uses them to cluster groups. This assumption is sensible because dimensions are typically machine-generated as is the case with the FIST data. Human-generated data may contain errors or inconsistencies in the dimension (categorical) attributes. Many existing systems (e.g., HoloClean [57], Raha [46]) leverage constraints and patterns to fix dimension errors, and we recommend these be run before Reptile.

Note: The paper strives to be self-contained. References to appendices can be skipped, and are in the technical report [4].

2 APPROACH OVERVIEW

This section presents our problem definition, and describes model-based repair and the role of auxiliary datasets. We then walk through the components in `Reptile`.

2.1 Problem Definition

Given relation $\mathbf{R}$ with attributes $\mathbf{A}$, we assume that all the attributes in $\mathbf{R}$ are partitioned into hierarchical dimensions. A dimension's hierarchy $H = [A_1, \ldots, A_k]$ is an ordered list of attributes where there is a functional dependency $A_n \rightarrow A_m \forall m < n$. We say that A_n is *more specific* than A_m if $m < n$ in the same hierarchy. The last attribute A_k is the most specific attribute in H. Note that a dimension's hierarchy may contain a single attribute.

User explores $\mathbf{R}$ through initial view $V = \gamma_{A_{gb}, f(A_{agg})}(\mathbf{R})$, where $A_{gb} \subset \mathbf{A}$, $A_{agg} \subset \mathbf{A}$, $A_{gb} \cap A_{agg} = \emptyset$, $f(\cdot)$ is a distributive [32] aggregation function such that, given the partition of $\mathbf{R}$ into J subsets $\mathbf{P}_1, \cdots, \mathbf{P}_J$, there exists function G: $f(\mathbf{R}) = G(f(\mathbf{P}_1), \ldots, f(\mathbf{P}_J))$. Let $t_i \in V$ be an output tuple and $t_i[agg]$ be t_i's aggregation result.

Example 2 (Distributive aggregation function). *Count is a distributive aggregation function. Given the partition of* $\mathbf{R}$ *into* J *subsets* $\mathbf{P}_1, \cdots, \mathbf{P}_J$, *we can find* G_{count} *such that* $count(\mathbf{R}) = G_{count}(\{count(\mathbf{P}_1), \cdots, count(\mathbf{P}_J)\}) = \sum_{i=1}^{J}(count(\mathbf{P}_i))$

User can make complaint about tuple $t_c \in V$. Define user complaint as a function $f_{comp} : t \rightarrow \mathbf{R}$ [2] which takes tuple as input and outputs a value that user aims to minimize. This formulation captures common complaints [11, 51, 59, 72], such as t[agg] is too high or too low, or that t[agg] should be a specific value. For instance, $f_{comp}(t) = |t[count] - v|$ states that the output attribute `count` should have been v.

We exploit the hierarchical structure to drill-down from the complaint tuple along different dimensions (e.g., district to village, or from year to month). Given a tuple t in the current view $V = \gamma_{A_{gb}, f(A_{agg})}(\mathbf{R})$, $drilldown(V, t, H)$ adds the next strict attribute in hierarchy H to A_{gb} in V and replaces $\mathbf{R}$ by the provenance of t.

Example 3 (Drill-down). *Figure 1 is grouped along geographic and temporal dimensions, with hierarchies* H_{geo}=[`District, Village`] *and* H_{time} = [`Year, Month`]. *Figure 1a shows the view* V *that filters and aggregates by* (`District=Ofla, Year`). *Let* t *be the tuple for year* `1986`. $drilldown(V, t, H_{geo})$ *would further aggregate the provenance of* t *by village (Figure 1b).*

Next, we apply repair-based intervention to the statistics of each tuple in drill-down result and check its effect on the complained tuple t'_c. Let $f_{repair} : t \rightarrow t$ be a repair function that, given a tuple, returns a tuple with its expected (repaired) aggregate statistics. We use f_{repair} to repair tuple in drill-down result, and efficiently compute its repair effect on t'_c through distributive property: given complaint tuple t_c, drill-down results $V' = drilldown(V, t_c, H)$ and subgroup $t' \in V'$, repairing $t' \rightarrow f_{repair}(t')$ updates the complaint to $t'_c = G(V' - \{t'\} \cup \{f_{repair}(t')\})$.

Finally, we search for one hierarchy $H \in \mathbf{H}$ to drill-down, and one tuple $t \in drilldown(V, t_c, H)$ such that "fixing" the t's *group statistics* would minimize user complaint $f_{comp}(t'_c)$.

[2] In general, f_{comp} may be an expression composed of distributive aggregates in the query. For instance, `SUM` can be decomposed into an expression over `MEAN` and `COUNT`

Problem 1 (Complaint-based Drill-down). *Given* t_c, f_{comp}, f_{repair}, *return the next drill-down hierarchy and tuple* (H^*, t^*) *where:*

$$H^*, t^* = \underset{H,t}{\arg\min} \quad f_{comp}(t'_c) \tag{1}$$

$$s.t. \quad V' = \text{drilldown}(V, t_c, H), \tag{2}$$

$$t'_c = G(V'/\{t\} \cup \{f_{repair}(t)\}) \tag{3}$$

$$H \in \mathbf{H} \tag{4}$$

$$t \in V' \tag{5}$$

Example 4 (Complaint-based Drill-down). *Given the complaint* t_c = (Year : 1986, District : Ofla, count : 62) *in Figure 1, the complaint function is* $f_{comp}(\text{count}) = |\text{count} - 70|$ *(that is, the count of tuples in Ofla in year 1986 should have been 70) and consider the* Darube *and* Zata *records after drilling down along* H_{geo} *(Figure 1b). If the repair function fixes* Darube*'s count to* 15, t_c*'s count will update to* 67*, and the complaint function returns* $f_{comp}(67) = 3$. *In contrast, if* Zata *is repaired to 16, then its complaint function would return* 1*, which is preferable. After searching all hierarchies, the top ranked (hierarchy, tuple) pair is returned.*

Although the user can easily provide t_c and f_{comp}, the repair function f_{repair} is hard for users to express (since they only see the aggregated effects of the errors), yet is critical to the problem. We describe how we address this issue with *model-based repair* next.

2.2 Model-based Repair

Reptile identifies erroneous groups by comparing their statistics with expected statistics based on a model. Models are commonly used to identify and repair numeric errors, and prior works have used log-linear [63] and linear regression models [51]. Models provide the flexibility to combine features derived from the drill-down groups, as well as from auxiliary datasets (e.g., satellite sensing data in Example 1). However, sparsity is the major challenge, as there may not be enough drill-down results (e.g., villages in Ofla in 1998) to fit an accurate model. We now describe how we address this challenge, and how users can tune the repair function.

2.2.1 *The Repair Model.*

A Naive Approach is to use the results of a candidate drill-down as the training examples for a linear regression model $\boldsymbol{y} = \boldsymbol{X} \cdot \boldsymbol{\beta} + \boldsymbol{\varepsilon}$. For instance, after drilling down from Ofla to its villages, $\boldsymbol{y}$ is the result of the complaint's aggregation function $f(\cdot)$[3] for each village, and $\boldsymbol{X}$ is the feature matrix derived from village-level information (e.g., population, crops, rainfalls). The main problem is that Ofla alone may not contain enough villages to train an accurate model.

Parallel Groups: Reptile uses the drill-down results of all parallel groups (e.g., villages from other districts and years). In the FIST example, there are 34 years and 295 villages, and using parallel groups increases the number of training examples to 10030.

Multi-level Models: The dataset's hierarchical structure naturally clusters the drill-down groups: villages in the arid Tigray region will be dissimilar from villages in the tropical Harari region. This effect is common in fields such as sociology [26], demography [61], public health [23], and market sectors [70]. Unfortunately, linear models do not take this hierarchical structure into account. As compared to linear models, we show that multi-level models increase Reptile's top-3 accuracy by >15% (Section 5.2).

Reptile uses multi-level linear models by default. Multi-level models fit a set of global parameters, as well as separate parameters for each parent group (e.g., year, district)—termed "clusters" for convenience—to account for their variations. Suppose we are drilling down from clusters defined by A_{gb} (e.g., year, district) to A'_{gb} (e.g., year ,district, village), and there are $\mathcal{G}$ clusters. The model for the i^{th} cluster is defined as:

$$\boldsymbol{y}_i = \boldsymbol{X}_i \cdot \boldsymbol{\beta} + \boldsymbol{Z}_i \cdot \mathbf{b}_i + \boldsymbol{\varepsilon}_i, i = 1, ..., \mathcal{G} \tag{6}$$
$$\mathbf{b}_i \sim \mathcal{N}(\boldsymbol{0}, \Sigma), \boldsymbol{\varepsilon}_i \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$$

where $\boldsymbol{y}_i$ is the vector of distributive aggregation $f(\cdot)$ result, and $\boldsymbol{X}_i$ is the feature matrix. The key difference from the linear model is the additional term $\boldsymbol{Z}_i \cdot \mathbf{b}_i$, which encodes random effects that vary across clusters. It can be interpreted as modeling the residual after fitting the global parameters in $\boldsymbol{X}_i \cdot \boldsymbol{\beta}$. $\boldsymbol{Z}_i$ is the random effects, set to $\boldsymbol{X}_i$ by default; $\mathbf{b}_i$ is the cluster-specific parameter drawn from a gaussian. Since $\boldsymbol{X}_i$ contains e.g., village, district, and year level attributes, $\boldsymbol{Z}_i$ may be tuned to only keep attributes relevant within clusters. $\boldsymbol{\varepsilon}_i$ is a cluster-specific error; $\boldsymbol{I}$ is the identity; $\boldsymbol{\beta}$, $\boldsymbol{\Sigma}$ and σ are parameters. Finally, the full model is constructed by (logically) concatenating each cluster's matrices.

Multilevel models exploit effects $\boldsymbol{\beta}$ shared among clusters (e.g. high rainfalls decrease drought across districts) that is common in hierarchical data, and uses $\mathbf{b}_i$ to fine-tune each cluster (e.g. how rainfall uniquely affected a district). If the effects are random across clusters (e.g. high rainfall affects districts randomly), the multilevel model reduces to an independent linear model per cluster.

By default, Reptile adopts multi-level *linear* models because linear models are simple and multi-level models are well-regarded in many scientific domains that study hierarchical data. In principle, Reptile supports user-defined models, which will be used to replace f_{repair} in problem definition.

2.2.2 *Tuning the Repair Function.*

Reptile provides many ways to improve the accuracy of the repair function, such as providing auxiliary datasets, custom featurizations, and customizing the random-effect matrix. For space constraints, we describe Reptile's default features and the role of auxiliary datasets, and defer the others to the technical report [4].

Default Features: Reptile treats all non-aggregated attributes in the drill-down results as categorical. However, naive hot-one featurization exacerbates the sparsity problem. Inspired by multivariate anomaly detection [39] and OLAP data cubes [63], we featurize attributes based on their *main effects* [47] to capture the similarity of statistics between clusters (e.g. the average drought severity should be similar between neighbour districts). To featurize an un-cleaned dataset, Reptile assumes that the majority of statistics are clean and replaces each categorical attribute value with the median of cluster statistics $\boldsymbol{Y}$. This is robust when less than 50% of values are corrupt [34]. For instance, if we drill-down to (district, village, year) and compute the per-cluster average, then the feature for 1985 would be the median of the average severities across all villages in 1985.

[3]In general, the complaint's aggregate can be composed of multiple distributive aggregates. In this case, we fit separate models for the distributive aggregates.

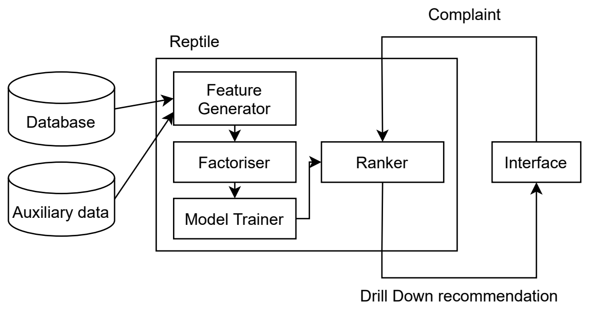


Figure 2: `Reptile` architecture

Note that the default features and the multi-level model exploit different data properties. The default features capture statistical similarities across dimensions, and the weights of these features are reduced if the similarities are not significant. The multi-level model captures similarity between the effects of features—if a feature is correlated with drought in one district, it should be correlated in other districts as well. The multi-level model is data-efficient by sharing weights between clusters, yet can fine-tune individual clusters.

Auxiliary Datasets: When possible, `Reptile` automatically joins auxiliary datasets with the drill-down results and includes their measures in the feature matrix. For instance, the village rainfall data in Example 1 is included once `Reptile` drills down to village. Users simply specify the dataset, join conditions, and measures. Since the user provides the auxiliary dataset, we assume it is clean. If auxiliary dataset is noisy and not predictive, the model will simply lower its feature weights, which can be a signal to check the auxiliary data. Section 5.2 shows that `Reptile` can benefit from the auxiliary data even when it's slightly correlated with the underlying clean data.

2.3 Usage Walkthrough and Architecture

We illustrate how the Financial Instruments Sector Team (FIST) uses `Reptile` to solve complaint-based drill-down problem (Example 1). `Reptile` is initialized with the database and attribute hierarchies (e.g, geo and temporal). A FIST researcher studies the annual severities in the Ofla district. She suspects that the standard deviation in 1986 is too high and submits it as a complaint. She also provides village-level rainfall as an auxiliary joined dataset because she feels it can help indicate droughts.

At this point, `Reptile` follows the architecture in Figure 2. It first combines the queried tables with the auxiliary sensing dataset, and uses them to extract model features. The *Factoriser* stores the features in an efficient factorised representation (described below), and the *Model Trainer* fits a predictive model to estimate the statistics for each group in the next candidate drill-down. For instance, if `Reptile` drills down along geography, the model estimates village level statistics in Ofla 1986. `Reptile` uses multi-level models to account for hierarchical relationships, and introduces optimized matrix operations over factorised representations.

The *Ranker* first evaluates each group (e.g., village) based on the extent that repairing the group's statistics to its expected value would address the complaint; it returns the top across every drill-down hierarchy. The researcher can examine the recommended groups, and optionally submit a new complaint at the village level to continue "zooming-in".

2.4 Extensions

We discuss simple extensions that further improve usability.

Set of Distributive Aggregations: The problem assumes that $f(\cdot)$ is a distributive aggregation function like `COUNT` for simplicity. However, $f(\cdot)$ can be extended to a general distributive set of functions like {`MEAN`, `COUNT`, `STD`} as discussed in Appendix A. Distributive set of functions is highly expressive and covers most of the aggregates studied in data science. For example, the percentage of obesity and population in public health [50], the frequency of physical activity in sociology [13]; and product innovation statitics in economics [40] can all be expressed as distributive set of functions.

Multi-attribute and multi-dimension drill down: The formal problem focuses on drill-down along a single dimension and attribute in each iteration. However, the user can also specify a set of "coupled" attributes (potentially from different hierarchies) that they want to drill-down all at once. To do so, `Reptile` constructs a special hierarchy H^* where the next attribute to drill down is a composite of the coupled attributes. All other steps and optimizations apply. Naturally, `Reptile` relies on the user to supply the coupled attributes, since the set is combinatorial.

Early Stopping: For dataset with too many hierarchies, drilling down all of them will be slow and the final group after many iterations may be too sparse to interpret. In practice, we find that FIST researchers naturally stop drilling down when satisfied, or when the group is small enough to inspect the raw data. Philosophically, we don't recommend a fully automated early stopping process, as cleaning should still be user-guided.

3 FACTORISATION BACKGROUND

Given the generated features, directly joining them together will result in a large feature matrix that exhibits vast redundancy. `Reptile` generates and trains over a factorised representation of the feature matrix to remove the redundancy, which decreases space and training time. This section provides background on factorised representations of join-aggregation queries and feature matrices. Readers familiar with these topics may jump to the next section.

3.1 Factorised Representations

Joins and hierarchical data exhibit redundancy when encoded in a tabular format, and factorised representations [54] (f-representations) remove this redundancy. Assuming a fully normalized database (e.g., in BCNF), f-representations encode query results as an algebraic expression composed of unions and cartesian products. In a join query, for instance, the set of left and right records that have the same join key will emit the logical cartesian product and f-representations avoid materializing it. Matrix operations in model training reduce to batches of aggregations [64], which can be efficiently executed over f-representations by pushing them through joins.

Given a relational table with schema S, the following notations are used for f-representations:

- $\{(v) : i\}$: a unary relation with tuple (v) whose count is i .
- $(E_1 \cup ... \cup E_n)$: union of relations $E_1, ..., E_n$ with the same schema.
- $(E_1 \times ... \times E_n)$: cartesian product of relations $E_1, ..., E_n$, where the schema of E_i is S_i and $S_1 \cap ... \cap S_n = \emptyset$.

F-representations remove redundancies due to functional dependencies inside a hierarchy and independence between hierarchies:

Example 5 (Hierarchical Data). *Consider the relation* $R = \{(a_1, b_1) : 1, (a_1, b_2) : 1, (a_2, b_3) : 1, (a_2, b_4) : 1\}$ *over schema* $S = [A, B]$, *with functional dependency* $B \rightarrow A$. *Its f-representation is:*

$$(\{(a_1) : 1\} \times (\{(b_1) : 1\} \cup \{(b_2) : 1\})) \cup (\{(a_2) : 1\} \times (\{(b_3) : 1\} \cup \{(b_4) : 1\}))$$

Example 6 (Independent Schemas). *Consider relation* $R_1 = \{(a_1) : 1, (a_2) : 1, (a_3) : 1\}$ *over schema* $S_1 = [A]$ *and relation* $R_2 = \{(b_1) : 1, (b_2) : 1, (b_3) : 1\}$ *over schema* $S_2 = [B]$. *Their schemas do not overlap, so the join result is quadratic in size (i.e., 9), whereas its f-representation is linear:*

$$(\{(a_1) : 1\} \cup \{(a_2) : 1\} \cup \{(a_3) : 1\}) \times (\{(b_1) : 1\} \cup \{(b_2) : 1\} \cup \{(b_3) : 1\})$$

Reptile develops matrix operations over f-representations, which are decomposed into batches of aggregation queries (Section 4.2). Here, we introduce the aggregation operator using a COUNT-query example and describe when they can be logically pushed through joins. For further background, please refer to [52].

Consider the aggregation $\gamma_{X_1,...,X_f,COUNT}(R_1 \bowtie ... \bowtie R_n)$, where the schema of the join result is $X_1, ..., X_f, X_{f+1}, ..., X_m$. For tuple t in relation R, the notation R[t] returns the COUNT for tuple t. Then, the aggregation result is:

$$Q[(X_1, ..., X_f)] = \bigoplus_{X_{f+1}} \cdots \bigoplus_{X_m} \bigotimes_{i \in [n]} R_i[S_i]$$

where $\bigotimes$ is the join subplan, $\bigoplus_X$ is an aggregation that marginalizes over attribute X, and S_i is the schema of relation R_i. $\bigotimes$ and $\bigoplus_X$ are defined as:

$$(R \bigotimes T)[t] = R[\pi_{S_1}(t)] * T[\pi_{S_2}(t)] \qquad \forall t \in D_1$$

$$(\bigoplus_X R)[t] = \sum \{R[t_1] | \ t_1 \in Dom(S_1), t = \pi_{S_1 \setminus \{X\}}(t_1)\} \qquad \forall t \in D_2$$

where S_1 and S_2 are the schemas for R and T, $X \in S_1$, $D_1 = Dom(S_1 \cup S_2)$, and $D_2 = Dom(S_1 \setminus \{X\})$. Suppose t = ⟨Distinct = Ofla⟩. The first statement says that Ofla's COUNT after the join is equivalent to multiplying the COUNT Ofla records in R and T. The second statement states that marginalizing over X (say, the attribute Year) is computed as the sum of Ofla counts over every year.

Example 7 (Join and Aggregation Operators). *Let relations* $R = \{(a_1, b_1) : 1, (a_2, b_1) : 2\}$ *over schema* $[A, B]$, *and relation* $T = \{(b_1, c_1) : 3, (b_1, c_2) : 4\}$ *over schema* $[B, C]$. *Consider the query:*

$$Q[(A, B)] = \bigoplus_C (R[(A, B)] \bigotimes T[(B, C)])$$

The intermediate result $R_{\bowtie} = R[(A, B)] \bigotimes T[(B, C)]$ *contains:*

$$\{(a_1, b_1, c_1) : 3, (a_1, b_1, c_2) : 4, (a_2, b_1, c_1) : 6, (a_2, b_1, c_2) : 8\}$$

$\bigoplus_C$ *partitions* $R_{\bowtie}$ *by A,B and sums all the counts in each partition to derive* $\{(a_1, b_1) : 7, (a_2, b_1) : 14\}$.

Early marginalization pushes $\bigoplus_C$ down when C is not used in the outer query (such as joins):

Example 8 (Early Marginalization). *Let relations* R, T *have schemas* $[A, B]$ *and* $[B, C]$. *Consider the query* $\gamma_{A,COUNT}(R \bowtie T)$, *where each attribute's domain is* $O(n)$. *Both relations are thus* $O(n^2)$ *and the join result is* $O(n^3)$. *Notice that attribute* C *is not used for the join and can be marginalized early:*

$$Q[(A)] = \bigoplus_B \bigoplus_C (R[(A, B)] \bigotimes T[(B, C)])$$

Thus $\bigoplus_C$ *can be pushed through* $\bigotimes$ *to reduce the join result to* $O(n^2)$*:*

$$Q[(A)] = \bigoplus_B R[(A, B)] \bigotimes (\bigoplus_C T[(B, C)])$$

3.2 Factorised Feature Matrix

We now discuss how to construct the feature matrix using the example in Figure 3. Rather than materialize the full matrix, we construct a factorised matrix representation[4] in the form of a tree, where each node is either an attribute value, union (∪), or cartesian product (×) (see Section 3.1). To do so, we must first assign an attribute ordering—matrices expect a fixed column order—that dictates the attributes encoded at each level in the f-representation.

Attribute Ordering: We order the attributes by selecting an ordering of the hierarchies, and within each hierarchy, order the attributes from least to most specific. The specific hierarchy order has no impact on performance, since the f-representation of the matrix can be efficiently translated into a different ordering during matrix multiplications. The main restriction is that the hierarchy that we are drilling down should be ordered last. Note that drilling down along different hierarchies will necessitate different attribute orderings; we describe work-sharing optimizations in Section 4.4.

Figure 3a shows data from two hierarchies: Time with attribute T and Geo with attributes District (D) and Village (V). Suppose the hierarchy ordering is [Time, Geo]. The fully materialized matrix $\boldsymbol{X}$ (Figure 3b) is computed as the cross product between the two hierarchy tables. Note the redundancy across the hierarchies (t_i is replicated), and within the Geo hierarchy (d_1 is replicated).

Factorised Feature Matrix: We now outline the construction of the factorised feature matrix using Figure 3c as the example. We refer readers to Olteanu et al. [54] for a complete procedure[5]. Each attribute corresponds to one level of the tree, and A_i's level is directly above A_j's if A_i directly precedes A_j in the attribute order. Each node (e.g., t_1) in a level corresponds to a distinct attribute value, and levels are connected via operators × and ∪. The edge structure between levels is dictated by whether the attributes are within the same hierarchy or not. In Figure 3c, *Time* directly precedes *District* but is in a separate hierarchy, thus the *District* nodes are unioned (∪) and connected to the *Time* level with (×). In contrast, attributes within the same hierarchy form a tree structure because villages are strictly partitioned by their district—the example removes redundant instances of t_1, t_2, and d_1.

Matrix operations iterate through the matrix row- or column-wise. Row-wise iterations requires decomposed aggregates (Section 4.2) to exploit redundancy, while column-wise iterations correspond to efficient Depth-First traversals through the f-representation's tree structure. Appendix C describes the detailed implementation.

[4]For legibility, we use attribute and feature interchangeably. See Appendix B for details.
[5]In their parlance, our "f-tree" does not contain branches.

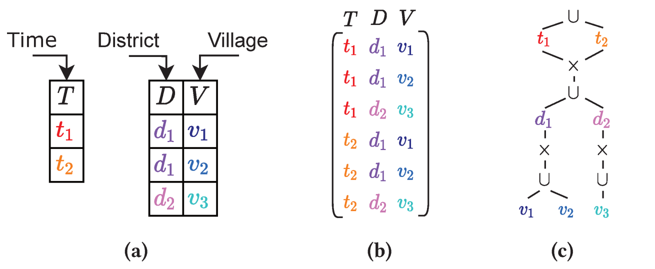


Figure 3: Example dataset with (a) Attribute values organized by hierarchy in attribute order, (b) materialized feature matrix, (c) factorised feature matrix.

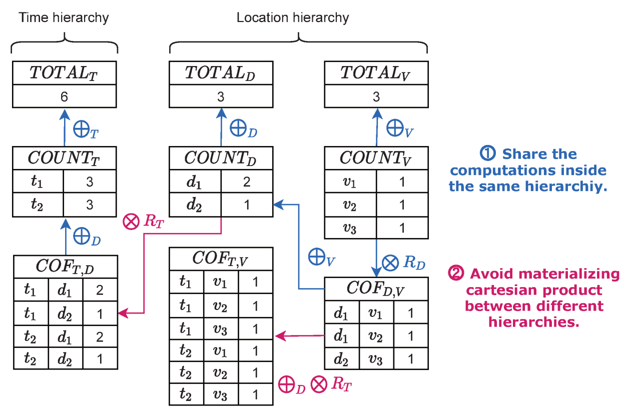


Figure 4: Aggregation results and Multi-query execution.

4 DETAILS AND OPTIMIZATIONS

In each iteration, Reptile recommends the next drill-down hierarchy and returns the top ranked groups (output tuples of the drill-down query). For each drill-down hierarchy, Reptile builds the factorised feature matrix, fits the multi-level model, estimates the expected statistics for each group using the model, and finally ranks the groups by their repair's effects on the user complaint. In this process, model training over factorised matrix is the primary bottleneck.

Prior works [64, 65] assume that the feature matrix is derived from a join query (without aggregation), where $\boldsymbol{Y}$ is simply another input attribute (e.g., Village in Figure 3b). This only requires gram matrix computations over factorised matrices for training because the concatenation between $\boldsymbol{X}$ and $\boldsymbol{Y}$ can be efficiently factorised. In contrast, Reptile executes join-aggregation queries where the $\boldsymbol{Y}$ values are aggregate statistics (e.g., average severity) potentially unique for each of the exponential number of groups. In short, factorisation does not reduce their redundancy. This motivates our matrix operation extensions between factorised and non-factorised matrices, as well as our optimizations.

Next, we decompose matrix operations into groups of aggregation queries efficiently executable over f-representations. We then develop work-sharing and caching optimizations to accelerate individual and multi-model training.

4.1 EM-based Model Training

We fit the multi-level model's parameters via maximum likelihood estimation using expectation maximization (EM). EM is widely used to train multi-level models and implemented in statistical packages such as lme [56] in R and statsmodels [66] in Python. In addition, our techniques apply to other algorithms (e.g., Fisher scoring [7], iterative generalized least squares [31]).

The EM algorithm (listed in Appendix D) is composed of 3 types of matrix multiplications—gram matrix ($\boldsymbol{X}^T \cdot \boldsymbol{X}$), right multiplication ($\boldsymbol{X} \cdot \boldsymbol{A}$), left multiplication ($\boldsymbol{B} \cdot \boldsymbol{X}$)—along with their per-cluster counterparts: $\boldsymbol{X}_i^T \cdot \boldsymbol{X}_i$, $\boldsymbol{X}_i \cdot \boldsymbol{C}_i$, $\boldsymbol{D}_i \cdot \boldsymbol{X}_i$ for the i^{th} cluster. Where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}_i, \boldsymbol{D}_i$ are intermediate matrices, and $\boldsymbol{X}$ is the factorised matrix.

To compute these operations, one naive approach is to materialize the full $\boldsymbol{X}$ matrix and use existing matrix operator implementations, but the matrix can be very large. Instead, we wish to directly perform matrix operations on the f-representation. Note that the *outputs* of Gram matrix, right and left multiplication are materialized as matrices because there is no redundancy to exploit.

4.2 Factorised Matrix Operations

Prior work [22, 38, 65] decomposes factorised representation operations into a batch of aggregation queries that can be used to directly compute cells in the output matrix. We first review the set of decomposed aggregations, and then describe our implementation of left and right multiplication that leverages the data's hierarchical structure. We describe work-sharing based on early marginalization, and then describe our novel drill-down specific optimizations.

4.2.1 *Decomposed Aggregates.*

Let us define three classes of count aggregations, $TOTAL_{A_i}$, $COUNT_{A_i}$, COF_{A_i,A_j}, that will be used to define matrix operation outputs. Recall that the feature matrix orders the attributes $A_n, \ldots, A_1$ by hierarchy, and from least to most specific attribute within each hierarchy. Using the attribute order *Time (T)*, *District (D)*, *Village (V)* in the running example, Figure 4 illustrates the outputs of these aggregation queries and their algebraic relationships to each other.

$TOTAL_{A_i}$ marginalizes over all attributes to the right of A_i (in attribute order), inclusive; it returns a single count value. $COUNT_{A_i}$ marginalizes all attributes strictly to the right of A_i; it returns the count for every unique A_i value. COF_{A_i,A_j} groups by A_i and A_j and computes the count for each group. Formally:

$$TOTAL_{A_i} = \bigoplus_{A_1} \ldots \bigoplus_{A_i} \pi_{A_i}(R_i) \bigotimes_{i \in [i-1]} R_i$$

$$COUNT_{A_i} = \bigoplus_{A_1} \ldots \bigoplus_{A_{i-1}} \pi_{A_i}(R_i) \bigotimes_{i \in [i-1]} R_i$$

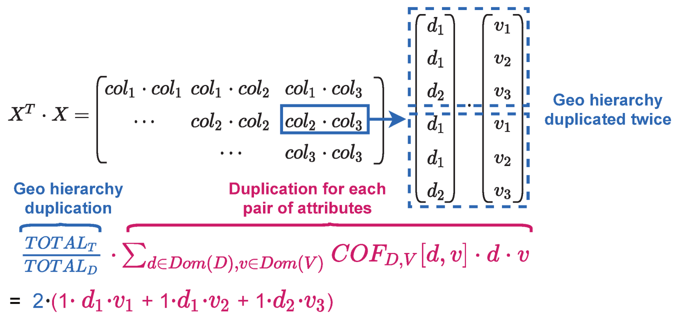
$$COF_{A_i,A_j} = \bigoplus_{A_1} \ldots \bigoplus_{A_{j-1}} \bigoplus_{A_{j+1}} \ldots \bigoplus_{A_{i-1}} \pi_{A_i}(R_i) \bigotimes_{i \in [i-1]} R_i$$

$$i \in [1, n], j \in [1, i-1]$$

Rather than execute these queries by joining then aggregating, we next describe a multi-query optimization to push down marginalization and minimize intermediate join sizes.

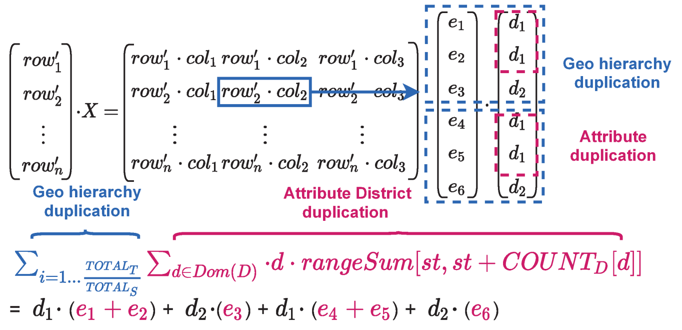
4.2.2 *Matrix Operations Using Decomposed Aggregates.*

We now present the intuition for optimizing the most expensive three matrix



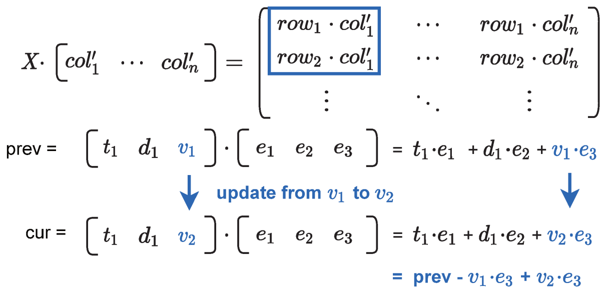
$X^T \cdot X =$
Geo hierarchy duplicated twice
Geo hierarchy duplication
Duplication for each pair of attributes
$\frac{TOTAL_T}{TOTAL_D} \cdot \sum_{d \in Dom(D), v \in Dom(V)} COF_{D,V}[d,v] \cdot d \cdot v$
$= 2 \cdot (1 \cdot d_1 \cdot v_1 + 1 \cdot d_1 \cdot v_2 + 1 \cdot d_2 \cdot v_3)$

(a) Gram Matrix



Geo hierarchy duplication
Attribute duplication
Geo hierarchy duplication
Attribute District duplication
$\sum_{i=1 \ldots \frac{TOTAL_T}{TOTAL_S}} \sum_{d \in Dom(D)} \cdot d \cdot rangeSum[st, st + COUNT_D[d]]$
$= d_1 \cdot (e_1 + e_2) + d_2 \cdot (e_3) + d_1 \cdot (e_4 + e_5) + d_2 \cdot (e_6)$

(b) Left Multiplication



prev = $[t_1 \; d_1 \; v_1] \cdot [e_1 \; e_2 \; e_3] = t_1 \cdot e_1 + d_1 \cdot e_2 + v_1 \cdot e_3$
update from v_1 to v_2
cur = $[t_1 \; d_1 \; v_2] \cdot [e_1 \; e_2 \; e_3] = t_1 \cdot e_1 + d_1 \cdot e_2 + v_2 \cdot e_3$
$=$ prev $- v_1 \cdot e_3 + v_2 \cdot e_3$

(c) Right Multiplication

Figure 5: Example matrix operations.

operations—gram matrix, left and right multiplication—using data from the running example (Figure 3). The key idea is to use the decomposed aggregates above to quantify the redundancy (i.e., duplication) in the vector dot product computations in each output matrix cell. We also describe the main optimization for their per-cluster variants, and defer details to Appendix E, and focus on the principles. We note that the gram matrix implementation is the same as in [65], but we introduce an optimization based on the data hierarchies.

Gram Matrix: Figure 5a illustrates the dot product between columns col_2 and col_3 in $\boldsymbol{X}$. Since there are two times t_1 and t_2, the district and village data is duplicated twice. Instead of recomputing them, we compute $\frac{TOTAL_T}{TOTAL_D}$ to infer the number of times $col_2 \cdot col_3$ is duplicated, and $COF_{D,V}$ to account for the number of times each pair of district, village values are duplicated. Our major optimization is to observe that $COF_{A,B}$ is simply a cartesian product that does not need to be materialized when A and B are independent. This is the case when A and B are from different hierarchies.

Left Multiplication: Figure 5b shows this between a materialized matrix and $\boldsymbol{X}$; let row'_2 contain elements $[e_1, \ldots, e_6]$. To compute $row'_2 \cdot col_2$, the outer summation iterates over the districts values twice, once for each Times value t_1, t_2. Within each iteration (e.g., t_1), each district value is multiplied by the sum of the corresponding elements in row'_2. For instance, d_1 is multiplied by e_1 and e_2, while d_2 is multiplied by e_3. Since row'_2 will be referenced for every column in $\boldsymbol{X}$, we preprocess row'_2 by computing the prefix sum, to allow for fast range summations (e.g., $rangeSum[0, 2] = e_1 + e_2$). *st* is used to keep track of the start position of row'_2, and is updated for each range summation.

Right Multiplication: This uses rows in $\boldsymbol{X}$ so cannot benefit from the techniques above. However, vertically adjacent rows in $\boldsymbol{X}$ have considerable overlap (Figure 5c). For instance, row_1 and row_2 only differ in the last value ($v_1 \rightarrow v_2$). Thus, $row_2 \cdot col'_1$ can be incrementally computed from the preceeding row's dot product result.

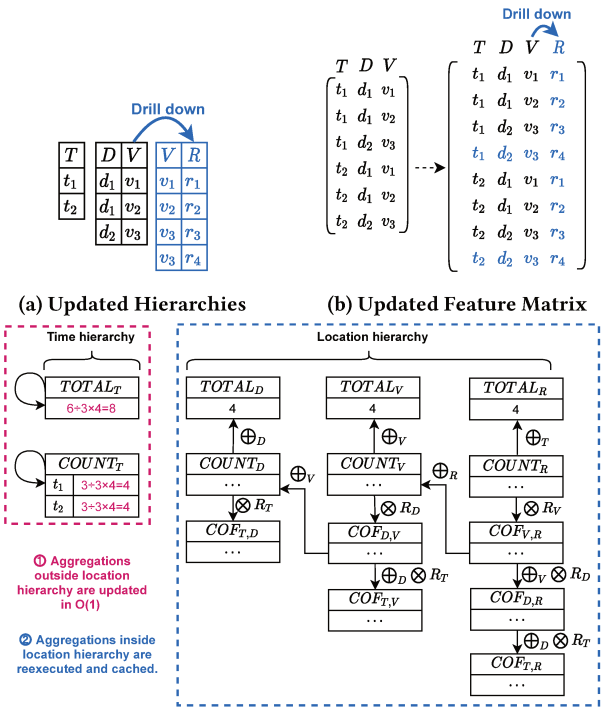
Per-cluster Optimizations: The per-cluster variants use the same algorithms over the clusters' sub-matrices. Since clusters correspond to siblings in the f-representation (e.g., districts d_1 and d_2 in Figure 3c), they are amenable to the same work sharing optimization as for right multiplication. For instance, the first cluster (rows 1, 2) and second cluster (row 3) in Figure 3c share t_1, and can cache t_1's contributes to the matrix operation's output.

4.3 Multi-Query Optimization

Early marginalization [65] pushes aggregation operators through joins, and work sharing computes decomposed aggregates *TOTAL*, *COUNT*, and *COF*. For instance, $TOTAL_D$ is simply the sum of counts in $COUNT_D$. Similarly, $COF_{D,V}$ can be computed as $COUNT_D \bigotimes R_V$, or as $COUNT_V \bigotimes R_D$. These relationships are depicted as edges in Figure 4. Given the dependency graph, the aggregations are simply computed in topological order. We use the same $COF_{A,B}$ optimization as described for gram matrix above, and avoid materializing the cartesian product for attributes from different hierarchies.

4.4 Drill-down Optimization

Equations 2 and 4 in the problem statement require drilling down each hierarchy, and each drill-down augments the factorised feature matrix with the additional columns corresponding to the next attribute in the hierarchy. For instance in Figure 6a, the user further drills down along the geography hierarchy from Village (V) to Road (R), which expands the feature matrix (Figure 6b). Notice that after drilling down to Road, the multiplicities of the preceding attributes change—t_1 is duplicated 4 rather than 3 times, and v_3 is duplicated twice for each r_i value. Although the decomposed aggregates for the attributes in the drill-down hierarchy (D, V, R) need to be recomputed (using the multi-query optimizations in the previous subsection), we can update each of the decomposed aggregates for attributes in the other hierarchies in O(1). For instance, the multiplicity for t_1 can be updated by dividing by the current $TOTAL_D$ (e.g., 3) and multiplying by the updated $TOTAL_D$ after the drill-down (e.g., 4). This is based on the observation that attributes between different hierarchies are independent. Figure 6c depicts the updated aggregates in the example.


Drill down
Time hierarchy
Location hierarchy
① Aggregations outside location hierarchy are updated in O(1)
② Aggregations inside location hierarchy are reexecuted and cached.

(a) Updated Hierarchies (b) Updated Feature Matrix

(c) Updated Aggregation Results

Figure 6: Example Updates after Drill-Down.

The user will ultimately pick one drill-down hierarchy that Reptile recommends (e.g., Time). However, the next call to Reptile would need to re-evaluate all hierarchies again, and we cache decomposed aggregates to accelerate this case.

Drilldown optimization differs from incremental view maintenance (IVM) for f-representations [52]. IVM updates query outputs assuming that the input update size O(Δ) is smaller than the relation size O(n). However, during drill-down, the decomposed aggregates of the other attributes all change due to the new attribute. Thus, $\Delta \approx n$ and does not benefit from IVM.

4.5 Putting It All Together

Reptile performs the following operations in each iteration to recommend the most promising drill-down results that will repair the user's complaint. For each candidate hierarchy H, it 1) constructs the factorised feature matrix after drill-down, 2) recomputes the decomposed aggregates for the attributes in H with multi-query optimizations, 3) updates each of the remaining decomposed aggregates in constant time, 4) translates EM into matrix operations that are executed until parameter convergence, 5) repairs each drill-down group based on the model prediction and incrementally updates the complaint to check the extent it is resolved.

5 EXPERIMENTS

We now evaluate the effectiveness of our optimizations, and assess Reptile's ability to identify group-wise data errors such as missing data or systematic corruptions. One challenge with proposing a new interactive cleaning method is the lack of existing benchmarks. Thus we evaluate runtimes using both synthetic data and complaints, and real-world case studies. The first case study is based on known and resolved errors in COVID-19 data, and the second is based on an expert user study with FIST data and team members.

Reptile is implemented in C++. All experiments are run single-threaded on a Macbook Pro with 1.4 GHz Quad-Core Intel Core i5, 8 GB 2133 Mhz LPDDDR3 memory, and 256GB SSD. All the experiments fit and run in memory.

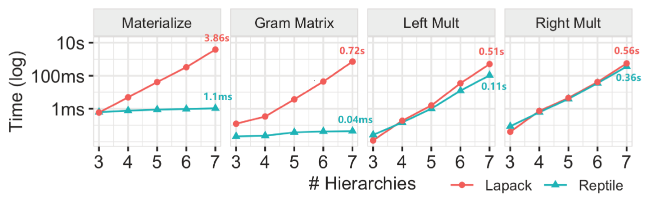


Figure 7: Matrix operation runtimes compared to Lapack-based implementation.

5.1 Performance Evaluation

Given a complaint, Reptile enumerates and drills down on each hierarchy, computes decomposed aggregates, builds the (factorised) feature matrix, trains the multi-level model, and ranks the groups. We first evaluate individual steps—the effectiveness of factorised matrix operations (Section 5.1.1), the cost of computing decomposed aggregates as compared to prior work (Section 5.1.2), and the drill-down optimizations (Section 5.1.3)—and then evaluate end-to-end run times on two real-world datasets (Section 5.1.4).

Default Setup: The attributes in the input relations are organized into hierarchies. The synthetic datasets vary the number of hierarchies (default: $d = 3$) and number of attributes in each hierarchy (default: $t = 3$). By default, each attribute contains $w = 10^6$ unique values. The data is in BCNF and sorted. Since we only report runtimes, we run Reptile to completion and return a random group.

5.1.1 Factorised Matrix Operations. The number of hierarchies d dictates the size of the feature matrix $\boldsymbol{X}$: exponential in the number of rows, and linear in the number of columns. Thus, the matrix materialization cost and gram matrix cost are exponential in d. In contrast,those of the factorised representation are linear in d.

We measure runtimes for matrix materialization, gram matrix, and left and right multiplication. The former compares the full and factorised matrix construction, while the latter three compare the Lapack[10] implementations over the full feature matrix with Reptile's factorised implementation. Lapack is a heavily optimized and widely used linear algebra library. To minimize the effects of our multi-query optimizations, each hierarchy is configured with only one attribute that has cardinality $w = 10$. Thus, the shape of $\boldsymbol{X}$ is $w^d \times t \cdot d = 10^d \times 3 \cdot d$.

Figure 7 reports runtimes in log scale. Materialization and gram matrix are exponential as a consequence of the matrix size, and the factorised implementations reduce the costs to linear. For left multiplication, we use a random 1×10^d matrix as input; the size of the random matrix dominates the cost, thus both methods increase

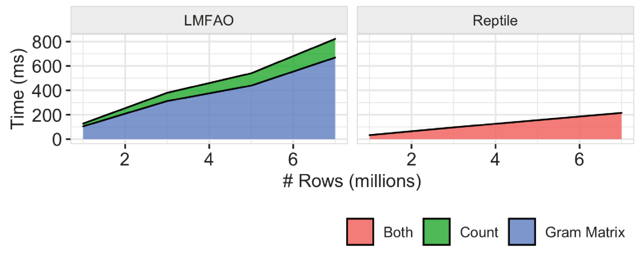


Figure 8: Multi-query execution

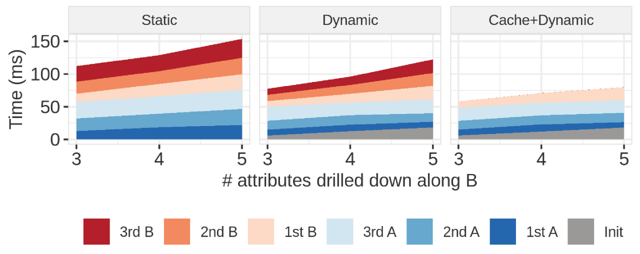


Figure 9: Drill-Down Optimization

exponentially. At 7 hierarchies, Reptile is 5× faster by exploiting redundancies in the matrix and the range sum optimization. For right multiplication, we use a random $3 \cdot d \times 1$ matrix. The runtime again grows exponentially due to the size of the output matrix (which is fully materialized due to the lack of inherent redundancy). At 7 hierarchies, Reptile is 1.6× faster by exploiting overlaps between vertically adjacent rows.

5.1.2 Multi-query execution. We now evaluate the benefits of our work-sharing multi-query optimizations for computing the decomposed aggregates *COUNT*, *COF*, and *TOTAL*. We compare against LMFAO[64], which is the state-of-art factorised batch aggregation engine implemented in C++ [64]. Its current implementation only supports computing *COUNT* and *COF* (as a by-product of computing the gram matrix), thus we use *COUNT* and gram matrix in the benchmark. In addition, LMFAO computes *COUNT* and the gram matrix serially, while Reptile shares their work, however this is simply an implementation detail that has minor benefits. *TOTAL* is quickly computed by scanning *COUNT* so we disregard it.

Since join cost is the bottleneck, we vary the cardinality for the attributes along the x-axis. Figure 8 shows that Reptile is over 4× faster than LMFAO. The primary reduction is due to our optimizations based on independence between hierarchies.

5.1.3 Drill-Down Optimization. We test the work-sharing of multi-query optimizations between multiple invocations of Reptile. Reptile uses hierarchy independence to update the non-drill-down hierarchies in constant time. Thus, two hierarchies, $A = [A_1, \ldots, A_6]$ and $B = [B_1, \ldots, B_6]$ are sufficient to characterize the drill-down costs and optimizations. In addition, the number of decomposed aggregates to compute is quadratic in the number of attributes that have already been drilled down upon.

For these reasons, we measure the cost of computing decomposed aggregates for each hierarchy by invoking Reptile three times, where we pick A to drill-down each time. We assume that for hierarchy A, we have already drilled down to A_3, and for hierarchy B, we have already drilled down $n = 3, 4, 5$ attributes. We compare Static, which recomputes decomposed aggregates for each query, Dynamic which exploits the independence between hierarchies, and Cache + Dynamic, which further reuses cached results from hierarchies not drilled down.

Figure 9 varies the number of attributes already drilled down along hierarchy B in the x-axis. For the areas, 3rdB means the cost of update hierarchy B's decomposed aggregates during the third invocation of Reptile. The gray area corresponds to the initial cost of computing the aggregates. The lines are stacked to show total runtimes to run Reptile. Dynamic is > 1.2× faster than Static by updating independent hierarchies more efficiently, while adding caching eliminates the cost of 2ndB and 3rdB, since their aggregates were computed and cached in the first Reptile invocation.

5.1.4 End-to-end Runtime Evaluation. Finally, we report end-to-end runtime using two popular real-world analysis datasets.

Absentee [3]: there are 179K records of North Carolina absentee voting data for 2020. We explore 4 hierarchies with one attribute each: county (100 unique values), party (6), week (53), gender (3). We invoke Reptile 4 times. Since we focus on runtime and not accuracy, we arbitrarily pick a sequence of drill-down attributes: county, party, week, gender.

COMPAS [2]: there are 60,843 records of defendant recidivism risk scores. We explore 4 hierarchies. Time hierarchy has 3 attributes (year, month and day; 704 unique days in total), and the remaining have one attribute each: age (3 ranges), race (6), and charge degree (3). We invoke Reptile 6 times, in the arbitrary drill-down attribute order: year, month, day, age range, race, charge degree.

For both datasets, the initial complaint is that the overall COUNT is too high, and the models are trained using 20 EM iterations. On the Absentee (COMPAS) dataset, Reptile took 1.6s (1.4s) while Lapack took 10.26s (8.8s); Reptile shows a 6× speedup over Lapack by not materializing and computing over the full feature matrix.

5.2 Explanation Accuracy: Synthetic Data

Reptile is unique in that it leverages complaints, hierarchical data, and multi-level models to identify group-wise data errors. We now evaluate and show the value of each of these design decisions via an ablation study, and also compare against two alternative approaches based on prior worsk. We use synthetic data to tune the problem difficulty and ensure a ground truth error.

5.2.1 Setup. In each Reptile invocation, the user picks the group statistic and Reptile selects top groups from the set of potential drill-downs. Thus, we designed the minimal experiment to evaluate how accurately Reptile can pick from the set of candidate drill-down groups. Since we evaluate each hierarchy independently, we vary the properties of dataset $\mathbb{R}$ with one hierarchy of 2 attributes $H = [A_1, A_2]$ and one measure for aggregation. We create 10 unique values for A_1, and 10 A_2 values for each each A_1 (e.g. 10 districts each with 10 villages). The count for each A_2 value (e.g. # reports per village) is drawn from a normal distribution $\mathcal{N}(100, 20)$, and each measure value is drawn from $\mathcal{N}(100 \times k_1, 20)$, where k_1 is a uniform random number in [0.5, 1.5] unique to each A_1 (e.g. different districts have different drought severity distributions). We impute errors to rows of three A_2 groups, and report the mean of 1000 datasets. For each approach, we return the top-3 A_2 values, and report accuracy.

Auxiliary Data: `Reptile` is able to combine auxiliary data (e.g. rainfall sensing data) provided by domain experts which has a (potentially weak) correlation to the correct aggregate statistics. We simulate this by generating one auxiliary dataset $\mathbb{R}_{aux}$ for each aggregate statistic (COUNT, MEAN, STD). The auxiliary table contains the same dimension attribute in $\mathbb{R}$, and one measure which is correlated ($\rho \in [0.6 - 1.0]$) with the aggregate statistic. To generate correlated random variables, we use the procedure proposed by Kaiser and Dickman [37], which applies linear transformation between orginal random variable and uncorrelated random variables from the same distribution. To reflect variance between A_1 groups (e.g., districts have different drought resistances), we multiply the auxiliary data by a uniform random number k_2 in [0.5, 1.5].

Error Generation: We introduce different classes of data errors: missing/duplicate records to change the COUNT statistics, and data drift [12] to change the MEAN statistic. To study the scenario where errors are distributed across dataset, we randomly choose three A_2 values and impute errors to their rows. For missing/duplicate, half of rows are deleted (`Missing`) or duplicated (`Dup`). For data drift, we either increase (↑) or decrease (↓) all measure values in the group by 5 to simulate a subtle systematic value error. We consider each error type individually, and in combination (**Missing** + ↓ and **Dup** + ↑). We submit COUNT and MEAN complaints for the individual COUNT and MEAN errors; we use SUM = MEAN × COUNT complaints for the combination errors.

Approaches: Eight approaches are used to identify the erroneous group: `Reptile`, `Linear`, `NoAux`, `Outlier`, `Raw-multi`, `Raw-truth`, `Sensitivity` [72] and `Support`. `Reptile` uses the auxiliary data and multi-level model. `Linear` uses a linear model. `NoAux` uses a multi-level model but not the auxiliary data. `Outlier` ignores the complaint and returns the group whose statistics most deviates from the model's prediction. Both `Raw` approaches use record-level repairs based on Winsorization [41]: they predict the mean and standard deviation of the measure for each drill-down group, and clip each row's measure to MEAN±STD. Then, they recommend groups that most resolve the complaint after clipping. `Raw-multi` uses the same multi-level model and auxiliary data in `Reptile` to predict MEAN and STD, while `Raw-truth` uses the ground truth's MEAN and STD (which is an upper-bound on this bottom-up approach).

Finally, we also compare with two prior explanation approaches: `Sensitivity` represents deletion-based interventions [72] and returns the group that best resolves the complaint if deleted. `Support` is a pruning criterion in scalable explanation systems like DIFF [5], and returns the group with highest COUNT (i.e., support) as there is only one dimension attribute.

5.2.2 Baselines. We first evaluate `Reptile` against `Linear`, `None-Aux`, `Raw-multi`, `Raw-truth`, and the two prior approaches (`Outlier` is defered next). Figure 10 varies the correlation of the auxiliary data (x-axis) for the different error types (columns). Both `Raw-multi` and `Raw-truth` completely fail to detect missing/duplicated records.

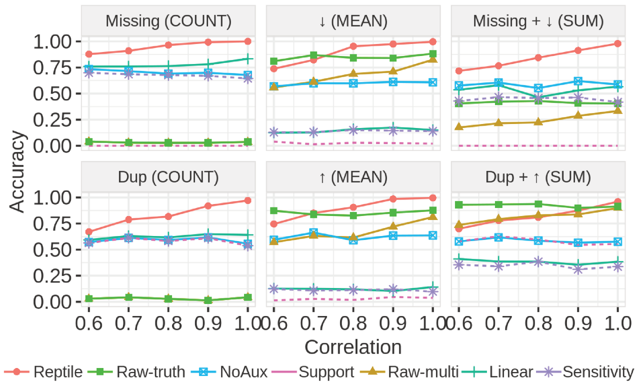


Figure 10: Accuracy comparison with naive approaches and prior work. ↑ is Increase, ↓ is Decrease, and Dup is Duplication. The complained aggregation is in the parentheses.

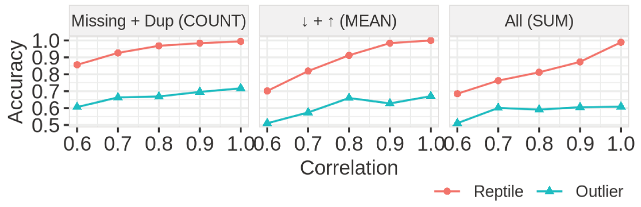


Figure 11: Accuracy comparison for multiple errors. `Outlier` doesn't take the complaint into account while `Reptile` does.

This is a limit of bottom-up approaches—they can't capture population errors, even given the ground truth. Both Raw approaches perform well for Duplication+Increase because they shift outlier values to the mean, which most affects (or is biased towards) SUM aggregations. The `linear` model fails to capture the variance between districts, and suggests the importance of multi-level models. `NoAux`, `Sensitivity` and `Support` are flat because they do not leverage auxiliary data. Finally, `Support` only performs well under duplication because it is density-based and is designed for "COUNT is high" complaints. `Reptile` is considerably and consistently more accurate, and successfully leverages the auxiliary data even when the correlation is weak.

5.2.3 Complaint Ablation. Real-world datasets often contain multiple errors in different subsets of the data [18]. We now show that complaints are critical to localize relevant errors as compared to traditional error detection approaches (we use `Outlier`) that don't use complaints. We generate three experimental conditions with different aggregation and error types. In each, we corrupt 3 groups, where only 2 groups affect the complaint and one group is a false positive. **Missing + Duplication**: two groups have missing records, and the false positive group has duplicates. The complaint is "COUNT is low". ↓ + ↑: two groups have data drift that decreases their measures values, and the false positive group with increased measure values. The complaint is "MEAN is low". **All:** two groups have decreased measure values *and* missing records, and the false-positive group has increased measure values and duplicates. The complaint is “SUM is low“.

Figure 11 shows that the complaint direction is critical to distinguishing between multiple error candidates. As expected, increasing the correlation of the auxiliary dataset helps better predict the true group statistics, however the accuracy of Outlier hovers around 0.66 because, while Outlier is able to identify three imputed groups, it cannot distinguish between them.

Takeaways: data errors manifest in a multitude of ways, however existing techniques are biased to specific complaint and/or error types. Reptile *supports general complaints as well as errors at the population level, and also benefits from auxiliary datasets if they provide signal (and does not degrade if they do not).*

5.3 Case Study: COVID-19

The COVID-19 data [24] maintained by the Johns Hopkins University Center for Systems Science and Engineering (JHU CSSE) contains two datasets. The US data contains 1,175,680 rows, location (state, county) and time (day) hierarchies, and count measures for confirmed infections and deaths. The global data contains 96,096 rows, location (country, state) and time (day) hierarchies, and measures for confirmed infections, deaths, and recoveries. Most statistics are reported at the country level (state is null), however large countries excluding the U.S. (e.g., Australia, Canada, China) report province/state-level statistics.

Setup: We use Github errors resolved between 12/2/2020 and 1/27/2021 as ground truth. We generate a corrupt dataset, submit a complaint for each issue, and compare methods. Most issues are due to missing data for a given day; others are due to backlogged reports across days n − m that are totaled and reported on day m + 1, or changes in a location’s reporting methodology [1]. We use 16 (14) issues from the US (global) datasets. For each, we filter by the complaint’s day, aggregate the total statistics at the parent geographical level (e.g, New York→USA), and specify whether the result is too high or too low. For instance, Texas under-reported infections on 1/21/2021, thus the complaint is that the total US cases on that day is too low. We compare Reptile with two explanation approaches [30]: Sensitivity used in Scorpion [72] and Support (described in Section 5.2), and two SOTA cleaning systems: Holoclean [57] and Raha [46].

Holoclean repairs measures using a probabilistic factor model. We used its default settings, where for each cell, Holoclean used the initial value, cell value frequency and co-occurence with other attributes as features to identify the most likely cell values. We attempted to provide functional dependencies between locations, but they didn’t hold (e.g., multiple states have counties with the same name). We found that Holoclean predicted the initial measures as most likely and thus didn’t repair any measures, so we used Holoclean to recommend groups with the least likely measures (on average). Raha is supervised, and uses manual labels to estimate measure likelihoods, so we used the ground truth data to generate labels. By default, Raha asked for 20 labels for training, but failed to identify any corrupted tuples because the model predicted 100% likelihood for every measure. We thus provided 300 labels and found that Raha mainly relies on its outlier, pattern violation, and

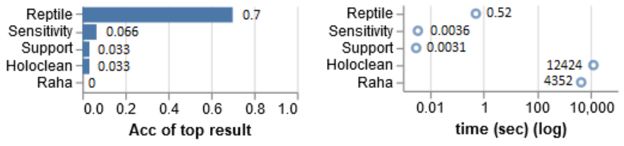


Figure 12: COVID-19 Case Study: Existing approaches either ignore complaints or fail to utilize expected statistics

rule violation detectors. Similar to Holoclean, we used Raha to recommend subgroups with the lowest average likelihood.

Results: Reptile took ≈0.5s to fit the model and recommend the group, while Sensitivity and Support took a few milliseconds as they only need to scan through the measures. Holoclean [57] took >3 hours and Raha took >1 hour to extract features and train their models. Figure 12 shows that Reptile is more accurate (70%) than the baselines (0 – 6.6%). Since the complaint was that the case count is too high, Sensitivity and Support simply returned states with the highest case counts, even though that was expected because those states had high populations. Both Holoclean and Raha failed because they ignore complaints and are designed for categorical attributes. Holoclean discretized the measures and encoded each domain value as a boolean random variable in the factor graph model, which couldn’t support numeric signal. Each cell’s features were also biased towards areas whose measures were stable over time, and these features were irrelevant to errors in the COVID-19 data. Raha attempted to generalize the labeled examples based on detector-generated features. Although Raha also considered column-wise Gaussian distributions, the true errors were caused mostly by deviation from individual expected statistics not column-wise averages.

Error Analysis: We conducted an error analysis of the 9 errors that Reptile did not identify. 5 issues were due to minor data drift across several weeks (e.g. a missing data source) that is later fixed. For instance, Quebec’s death statistics between 3/17/2020 to 1/27/2021 were all increased by a small amount, however the date range affect all Quebec data in the experiment. 4 issues were subtle issues smaller than the natural variation in the data, and won’t result in complaints. For example, on 12/18/2020, Washinton state submitted 21,308 instead of 21,038. See Appendix L for details.

Takeaways: although current cleaning systems are effective for categorical data, numerical data usually require custom, domain-specific features [17, 49] that are hard to automatically learn/infer. For instance, ERACER [49] repairs birth dates in genealogy databases using the parents’ birth dates, which does not generalize to the COVID-19 data. In contrast, Reptile *is generic, simply exploits hierarchical structure, and uses auxiliary data to encode domain expertise—these are readily available and widely utilized in many fields.*

5.4 Case Study: FIST

The Columbia University Financial Instruments Sector Team (FIST) group collects Ethiopian farmer-reported drought data to design drought insurance [55]. The data contains geography (Region, District, Village) and time (Year) hierarchies, and a severity measure from 1 (low severity) to 10 (high). The FIST group historically performed manual data cleaning based on domain expertise and by cross-referencing (noisy) external data sources (e.g., satellite estimates). We recruited 3 FIST team members[6] to use the system to submit complaints based on their experience, help verify the correctness of the results, and provide qualitative feedback (see Appendix M for screenshots, protocol, and further details). **Overall, Reptile correctly identified errors for 20 out of 22 complaints.**

Protocol and Complaints: Users are shown visualizations of annual Region-level statistics (count, mean, standard deviation). They click on suspicious statistics to create a complaint. Reptile recommends drill-downs and highlights the candidate group in the drill-down results. They can continue this process until they examine individual records to conclude whether the recommendations were correct. We ask users to follow a think-aloud protocol and share their interpretation throughout the cleaning process. Example complaints (and their rationale) include: "the MEAN in Tigray 2009 should be much higher because I remember farmers argued about this year (P1)", and "the STD in Medebay Zana 2018 is too high compared to other years (P2)".

Results and Failure Analysis: The users accepted 20 out of the 22 submitted complaints. These errors revealed issues such as: farmers that confuse planting and harvesting years (e.g. plant in one year, but harvest in the next year), misremember the events, report non-drought years as highly severe, and more. One failed complaint was due to inherent ambiguity and team members disagreed about the causes. The second was because a unique combination of two districts needed to be fixed together, but Reptile only return one of the two. See more details in Appendix M.

Qualitative Results and Discussion: FIST users said that Reptile "is valuable to clean and make sense of this massive data (P3)", "is helping to save the day for the project in Ethiopia during this year of Covid and civil strife (P1)." A major benefit is to automate group-level inspection and cross-reference with external data sources. P3 stated that "previously, we only had 5 villages in the Amhara region ... and data is cleaned manually using excel spreadsheet ... Now the project has scaled and we have 173 villages in Amhara. It is not possible to visit all these villages (P3)." Finally, users suggested that "it would be great [to] understand why the model makes certain prediction (P1)," and "I hope there are more flexible visualizations that display different satellite data in one geographic map (P2)."

6 RELATED WORK

Error Detection: Error detection traditionally uses integrity constraints [19] to find violations, while quantitative error detection often relies on statistical methods (e.g., outlier detection [11, 35, 43, 58] or explicit error-prediction models [33, 44, 46]). Reptile combines a complaint-based approach [5, 15, 51, 59, 72] based on how detected errors affect output complaints, with a model-based error prediction approach to identify candidate repairs.

Data Repair: Data repair is an optimization problem that satisfies a set of violated constraints over the database instance [19, 73], and can leverage signals (e.g., past repairs [71], knowledge bases [20]). Model-based repairs estimate the correct value of an error. ERACER [49] uses graphical models to repair raw data tuples. Active learning approaches [45, 69, 74] ask users to verify candidate repairs. Daisy [29] uses categorical histograms to identify and repair errors in join attributes. In Reptile, the user submits a single complaint over an aggregate query result, and the system trains multi-level models for aggregation-level repair. Finally, techniques such as unknown unknows [21] can be viewed as repairing group-wise missing record errors under species estimation assumptions.

Complaint-based Explanation: This class of problems follows the framework where, given a complaint over query results, they search for a good explanation from a candidate set (e.g., predicates, tuples, etc). They primarily differ in the ranking metric (e.g., sensitivity-based [5, 15, 59, 72], density-based [36, 60, 62], counterbalance [51]), and typically focus on deletion-based interventions. Reptile ranks drill-down groups based on aggregation level intervention which enables Reptile to uncover a broader range of errors like missing records which previous metrics fail to detect.

The hierarchical density attribution problem [25, 60, 62] returns a set of non-overlapping subgroups that account for the largest mass of the total density. Reptile is designed for a single hierarchy and supports more complex aggregation functions beyond density.

Factorised Representation: Factorised Representation [54] reduces redundancies due to functional dependencies, and has been used to optimize model training (linear regression [65], decision tree [38] and Rk-mean [22]) over factorised matrices derived from join queries. Reptile extends prior work [64, 65] to matrices based on join-aggregation queries that exhibit fewer redundancies, supports extra operations including right and left multiplication, and further exploits the hierarchical structure for optimization.

7 CONCLUSIONS

Classic cleaning techinques focus on categorical data, or make strong assumptions about error types. However real-world errors are often numeric, and there are fewer tools that are simple and fit a domain expert's mental model. Reptile focuses on numeric errors in the context of hierarchical data. Users encode expertise as complaints, auxiliary datasets, and features, and Reptile helps iteratively identify and repair numeric errors. Reptile recommends drill-downs by intervening on group statistics and finding the group whose intervention most fixes the complaint. Reptile trains a model to estimate each group's expected statistics, intervenes by setting a group's statistic to the model's estimate. Our implementation leverages a factorised matrix representation, and we developed factorised matrix operations as well as optimizations that leverage the data's hierarchical structure. Our optimizations reduce end-to-end runtimes by over 6× as compared to a Matlab-based implementation. Reptile identified 21 out of 30 data errors in John Hopkin's COVID-19 data, and identified 20 out of 22 complaints in a user study with Columbia University's Financial Instruments Sector Team based on their data collected from Ethiopian farmers.

ACKNOWLEDGMENTS

This work was supported in part by NSF 1845638, 2008295, 2106197, 2103794, and Amazon and Google awards.

[6]Novice users are ill-suited because good complaints rely on domain expertise.

REFERENCES

[1] 2021. Arizona Department of Health Services COVID-19 Data Dashboard. https://www.azdhs.gov/preparedness/epidemiology-disease-control/infectious-disease-epidemiology/covid-19/dashboards/.

[2] 2021. COMPAS Recidivism Risk Score Data and Analysis. https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis.

[3] 2021. North Carolina State Board of Elections: Provisional and Absentee Data. https://www.ncsbe.gov/results-data/absentee-data.

[4] 2021. (Technical Report) Reptile: Aggregation-level Explanations for Hierarchical Data. https://www.dropbox.com/s/g1seclgckyma5cm/Reptile_tech_report.pdf?dl=0.

[5] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, et al. 2020. DIFF: a relational interface for large-scale data explanation. *The VLDB Journal* (2020), 1–26.

[6] Eleanor Ainy, Pierre Bourhis, Susan B Davidson, Daniel Deutch, and Tova Milo. 2016. PROX: Approximated Summarization of Data Provenance. In *Advances in database technology: proceedings. International Conference on Extending Database Technology*, Vol. 2016. NIH Public Access, 620.

[7] Murray Aitkin and Nicholas Longford. 1986. Statistical modelling issues in school effectiveness studies. *Journal of the Royal Statistical Society: Series A (General)* 149, 1 (1986), 1–26.

[8] Hirotogu Akaike. 1998. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*. Springer, 199–213.

[9] Omar AlOmeir, Eugenie Yujing Lai, Mostafa Milani, and Rachel Pottinger. 2021. Summarizing Provenance of Aggregate Query Results in Relational Databases. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1955–1960.

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[11] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 541–556.

[12] Jean Paul Barddal, Heitor Murilo Gomes, Fabrício Enembreck, and Bernhard Pfahringer. 2017. A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. *Journal of Systems and Software* 127 (2017), 278–294.

[13] Jizzo R Bosdriesz, Margot I Witvliet, Tommy LS Visscher, and Anton E Kunst. 2012. The influence of the macro-environment on physical activity: a multilevel analysis of 38 countries worldwide. *International Journal of Behavioral Nutrition and Physical Activity* 9, 1 (2012), 1–13.

[14] Kenneth P Burnham and David R Anderson. 2004. Multimodel inference: understanding AIC and BIC in model selection. *Sociological methods & research* 33, 2 (2004), 261–304.

[15] Anup Chalamalla, Ihab F Ilyas, Mourad Ouzzani, and Paolo Papotti. 2014. Descriptive and prescriptive data cleaning. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 445–456.

[16] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758* (2020).

[17] Fang Chu, Yizhou Wang, D Stott Parker, and Carlo Zaniolo. 2005. Data cleaning using belief propagation. In *Proceedings of the 2nd international workshop on Information quality in information systems*. 99–104.

[18] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data*. 2201–2206.

[19] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proceedings of the VLDB Endowment* 6, 13 (2013), 1498–1509.

[20] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: reliable data cleaning with knowledge bases and crowdsourcing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1952–1955.

[21] Yeounoh Chung, Michael Lind Mortensen, Carsten Binnig, and Tim Kraska. 2018. Estimating the impact of unknown unknowns on aggregate query results. *ACM Transactions on Database Systems (TODS)* 43, 1 (2018), 1–37.

[22] Ryan Curtin, Benjamin Moseley, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*. 2742–2752.

[23] Ana V Diez-Roux. 2000. Multilevel analysis in public health research. *Annual review of public health* 21, 1 (2000), 171–192.

[24] Ensheng Dong, Hongru Du, and Lauren Gardner. 2020. An interactive web-based dashboard to track COVID-19 in real time. *The Lancet infectious diseases* 20, 5 (2020), 533–534.

[25] Ronald Fagin, R. Guha, Ravi Kumar, J. Novak, D. Sivakumar, and A. Tomkins. 2005. Multi-structural databases. In *PODS '05*.

[26] Roberto M Fernandez and Jane C Kulik. 1981. A multilevel model of life satisfaction: Effects of individual characteristics and neighborhood composition. *American Sociological Review* (1981), 840–850.

[27] Anne C Frenzel, Reinhard Pekrun, and Thomas Goetz. 2007. Perceived learning environment and students' emotional experiences: A multilevel analysis of mathematics classrooms. *Learning and Instruction* 17, 5 (2007), 478–493.

[28] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press.

[29] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 805–815.

[30] Boris Glavic, Alexandra Meliou, Sudeepa Roy, et al. 2021. Trends in Explanations: Understanding and Debugging Data-driven Systems. *Foundations and Trends® in Databases* 11, 3 (2021), 226–318.

[31] Harvey Goldstein. 1986. Multilevel mixed linear model analysis using iterative generalized least squares. *Biometrika* 73, 1 (1986), 43–56.

[32] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[33] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[34] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* 25 (2008).

[35] Victoria Hodge and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial intelligence review* 22, 2 (2004), 85–126.

[36] Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. 2015. Smart drill-down: A new data exploration operator. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 8. NIH Public Access, 1928.

[37] Henry F Kaiser and Kern Dickman. 1962. Sample and population score matrices and sample correlation matrices from an arbitrary population correlation matrix. *Psychometrika* 27, 2 (1962), 179–182.

[38] Lukas Kobis. 2017. Learning Decision Trees over Factorized Joins.

[39] Jorma Laurikkala, Martti Juhola, Erna Kentala, N Lavrac, S Miksch, and B Kavsek. 2000. Informal identification of outliers in medical data. In *Fifth international workshop on intelligent data analysis in medicine and pharmacology*, Vol. 1. Citeseer, 20–24.

[40] Daniel Lederman. 2010. An international multilevel analysis of product innovation. *Journal of International Business Studies* 41, 4 (2010), 606–619.

[41] Donald Lien and N Balakrishnan. 2005. On regression analysis with data cleaning via trimming, winsorization, and dichotomization. *Communications in Statistics—Simulation and Computation®* 34, 4 (2005), 839–849.

[42] Song Lin and Donald E Brown. 2006. An outlier-based data association method for linking criminal incidents. *Decision Support Systems* 41, 3 (2006), 604–615.

[43] F. Liu, K. Ting, and Z. Zhou. 2008. Isolation Forest. *2008 Eighth IEEE International Conference on Data Mining* (2008), 413–422.

[44] Zifan Liu, Zhechun Zhou, and Theodoros Rekatsinas. 2020. Picket: Self-supervised Data Diagnostics for ML Pipelines. *arXiv preprint arXiv:2006.04730* (2020).

[45] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: effective error correction via a unified context representation and transfer learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1948–1961.

[46] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*. 865–882.

[47] Leonard A Marascuilo and Patricia L Busk. 1987. Loglinear models: A way to study main effects and interactions for multidimensional contingency tables with categorical data. *Journal of Counseling Psychology* 34, 4 (1987), 443.

[48] Zelda Mariet, Rachael Harding, Sam Madden, et al. 2016. Outlier detection in heterogeneous datasets using automatic tuple expansion. (2016).

[49] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. 2010. ERACER: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 75–86.

[50] Neil K Mehta and Virginia W Chang. 2008. Weight status and restaurant availability: a multilevel analysis. *American journal of preventive medicine* 34, 2 (2008), 127–133.

[51] Zhengjie Miao, Qitian Zeng, Boris Glavic, and Sudeepa Roy. 2019. Going beyond provenance: Explaining query answers with pattern-based counterbalances. In *Proceedings of the 2019 International Conference on Management of Data*. 485–502.

[52] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380.

[53] Chris North and B. Shneiderman. 2000. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI '00*.

[54] Dan Olteanu and Jakub Závodný. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015),

1–44.

[55] Daniel Osgood, Bristol Powell, Rahel Diro, Carlos Farah, Markus Enenkel, Molly E Brown, Greg Husak, S Lucille Blakeley, Laura Hoffman, and Jessica L McCarty. 2018. Farmer Perception, Recollection, and Remote Sensing in Weather Index Insurance: An Ethiopia Case Study. *Remote Sensing* 10, 12 (2018), 1887.

[56] R Core Team. 2013. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[57] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proceedings of the VLDB Endowment* 10, 11 (2017).

[58] P. Rousseeuw and M. Hubert. 2011. Robust statistics for outlier detection. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1 (2011).

[59] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1579–1590.

[60] Matthias Ruhl, Mukund Sundararajan, and Qiqi Yan. 2018. The cascading analysts algorithm. In *Proceedings of the 2018 International Conference on Management of Data*. 1083–1096.

[61] Joshua M Sacco and Neal Schmitt. 2005. A dynamic multilevel model of demographic diversity and misfit effects. *Journal of Applied Psychology* 90, 2 (2005), 203.

[62] Sunita Sarawagi. 1999. Explaining differences in multidimensional aggregates. In *VLDB*, Vol. 99. Citeseer, 7–10.

[63] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. 1998. Discovery-driven exploration of OLAP data cubes. In *International Conference on Extending Database Technology*. Springer, 168–182.

[64] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An engine for batches of group-by aggregates. *arXiv preprint arXiv:2008.08657* (2020).

[65] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.

[66] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[67] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. 2009. Classification of imbalanced data: A review. *International journal of pattern recognition and artificial intelligence* 23, 04 (2009), 687–719.

[68] Jan Sundquist, Marianne Malmström, and Sven-Erik Johansson. 1999. Cardiovascular risk factors and the neighbourhood environment: a multilevel analysis. *International journal of epidemiology* 28, 5 (1999), 841–845.

[69] Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, and Nan Tang. 2017. Uguide: User-guided discovery of fd-detectable errors. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1385–1397.

[70] Steven Van de Walle, Bram Steijn, and Sebastian Jilke. 2015. Extrinsic motivation, PSM and labour market characteristics: A multilevel model of public sector employment preference in 26 countries. *International Review of Administrative Sciences* 81, 4 (2015), 833–855.

[71] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J Miller. 2014. Continuous data cleaning. In *2014 IEEE 30th international conference on data engineering*. IEEE, 244–255.

[72] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proc. VLDB Endow.* 6, 8 (June 2013), 553–564. https://doi.org/10.14778/2536354.2536356

[73] Mohamed Yakout, Laure Berti-Équille, and Ahmed K Elmagarmid. 2013. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 553–564.

[74] Mohamed Yakout, Ahmed K Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F Ilyas. 2011. Guided Data Repair. *Proceedings of the VLDB Endowment* 4, 5 (2011).

A PROBLEM DEFINITION

Distributive Set of Functions `Reptile` supports complaint over the results of a distributive set of aggregation functons. We extend the definition of distributive function [32] to a set of functions. A set of I aggregation functions $\mathbb{F}_{\text{agg}} = \{\text{F}_{\text{agg}_1}, \cdots, \text{F}_{\text{agg}_I}\}$ is distributive if, given the partition of R into $\mathcal{J}$ subsets, and the aggregation results $\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_{\mathcal{J}})$ after applying $\mathbb{F}_{\text{agg}}$ to $\mathcal{J}$ subsets, there exists function G such that: $\mathbb{F}_{\text{agg}}(\text{R}) = G(\{\text{R}_1, \cdots, \text{R}_{\mathcal{J}}\})$

For example, consider the following distributive set of aggregation functions: Mean, Count and Standard deviation. Given a set of $\mathcal{J}$ aggregation results $\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_{\mathcal{J}})$, there exists function $G = \{G_{\text{mean}}, G_{\text{count}}, G_{\text{std}}\}$ such that:

$$G_{\text{mean}}(\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_{\mathcal{J}})) = \frac{\sum_{\text{j}=1}^{\mathcal{J}} \text{F}_{\text{count}}(\text{R}_j) \cdot \text{F}_{\text{mean}}(\text{R}_j)}{\sum_{\text{j}=1}^{\mathcal{J}} \text{F}_{\text{count}}(\text{R}_j)}$$

$$G_{\text{count}}(\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_{\mathcal{J}})) = \sum_{\text{j}=1}^{\mathcal{J}} \text{F}_{\text{count}}(\text{R}_j)$$

$$G_{\text{std}}(\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_{\mathcal{J}})) =$$

$$\sqrt{\frac{\sum_{\text{j}=1}^{\mathcal{J}}(\text{F}_{\text{count}}(\text{R}_j) - 1) \cdot \text{F}^2_{\text{std}}(\text{R}_j) + \sum_{\text{j}=1}^{\mathcal{J}} \text{F}_{\text{count}}(\text{R}_j) \cdot (G_{\text{mean}} - \text{F}_{\text{mean}}(\text{R}_j))^2}{G_{\text{count}} - 1}}$$

B FEATURE MATRIX

In this section, we discuss, given all registered features, how to build feature matrix.

Attribute matrix: We first define attribute matrix, which helps us build feature matrix. Attribute matrix is built from the query result $Q = \gamma_{\text{A}'_{\text{gb}}, \text{f}(\text{A}_{\text{agg}})}(\mathbb{R})$ projected out aggregation function f and ordered by the attribute order (the same as feature matrix in Section 3.2). In Figure 13, given hierarchies in Figure 13a with order: Time and Location, attribute matrix is shown in Figure 13c.

Feature matrix: We then discuss how to derive feature matrix from attribute matrix. For feature registered with attribute A, given current view $\text{V}' = \gamma_{\text{A}'_{\text{gb}}, \text{f}(\text{A}_{\text{agg}})}(\text{prov}(\text{t}_\text{c}))$, this feature is applicable if $A \in \text{A}'_{\text{gb}}$. Given all applicable features and attribute matrix, feature matrix $\boldsymbol{X}$ is derived by replacing each attribute value in attribute matrix with feature values. Continue with examples in Figure 13, all applicable features are shown in Figure 13b and feature matrix is shown in Figure 13d.

Optimization: As an optimization during model training, feature matrix is not directly used. Instead, we isolate attribute matrix from feature matrix in aggregation queries. Because the mapping between attribute and feature is one-to-one, we can computate aggregation queries over attribute matrix, and infer the aggregation queries over feature matrix by mapping the value from attribute to feature. For example, in Figure 13, suppose we want to compute the sum of feature F^a in feature matrix (whose result is $3\text{f}^\text{a}_{\text{t}_1} + 3\text{f}^\text{a}_{\text{t}_2}$). We can first compute the count of each value $COUNT_\text{T}$ for attribute Time (T) (whose result is $\{\text{t}_1 : 3, \text{t}_2 : 3\}$). Suppose $\text{f}^\text{a}(\cdot)$ maps attribute Time (T) to feature F^a, then the sum of feature F^a can be computed by $\sum_{\text{a} \in \text{Dom}(\text{T})} COUNT_\text{T}(\text{a}) \cdot \text{f}^\text{a}(\text{a}) = 3\text{f}^\text{a}_{\text{t}_1} + 3\text{f}^\text{a}_{\text{t}_2}$. The isolation of attribute from feature can simplify the problem and improve performance because attribute matrix is smaller than feature matrix.

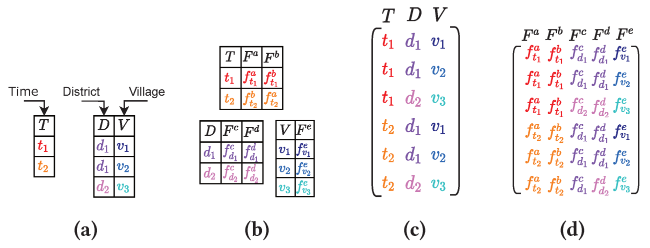


Figure 13: Example dataset with (a) hierarchies, (b) features, (c) attribute matrix, and (d) feature matrix.

C FACTORISER

In this section, we discuss the implementation and interface of *Factoriser* in `Reptile`. Given input relations, *Factoriser* in `Reptile` stores the factorised feature matrix, and presents an interface.

C.1 Storage

We discuss how *Factoriser* stores factorised feature matrix. *Factoriser* first exploits the one to one mapping between attribute value and feature value to store the factorised attribute matrix and the feature mapping separately. Then, to store factorised attribute matrix, `Reptile` doesn't materialize all unary relations and algebraic expressions in f-representation. Instead, `Reptile` exploits the independence between different hierarchies and functional dependencies inside each hierarchy, and stores factorised attribute matrix with relations implemented as sorted map. Sorted map makes it easy for `Reptile` to iterate through data. Given input relations, *Factoriser* normalizes relations to BCNF, sort them according to attribute order, and stores relations using sorted map. For example data in Figure 13 with attribute order "Time (T), District (D), City (V)", *Factoriser* stores normalized relations R[T] for Time hierarchy which enumerate attribute values, and R[D, V] for geography hierarchy which maps attribute District to Village. (R[D, V] is implemented as sorted map with key D and value V). *Factoriser* records the dependency among relations. Now consider the marginalization operation: $\bigoplus_\text{V} \text{R}[\text{D}, \text{V}]$. This marginalization is implemented by iterate through each District (key) and sum the count of its Villages (value).

C.2 Interface

Factoriser presents interface of relation and row iterator.

Relation: Given attribute, *Factoriser* returns relations, which are used to compute decomposed aggregates discussed in Section 4.2 to exploit redundancy in columns of attribute matrix. Given attribute A_i, if it is the least strict attribute in its hierarchy, factoriser returns $\text{R}_\text{i}[\text{A}_\text{i}]$ which enumerates all attribute values in A_i. Otherwise, factoriser returns $\text{R}_\text{i}[\text{A}_{\text{i}+1}, \text{A}_\text{i}]$ which connects A_i with the next less strict attribute $\text{A}_{\text{i}+1}$. For example, for example data in Figure 13, *Factoriser* returns $\text{R}_\text{T}[\text{T}]$, $\text{R}_\text{R}[\text{R}]$, and $\text{R}_\text{V}[\text{D}, \text{V}]$.

Row Iterator: For row iterator, *Factoriser* exploits the fact that the rows in attribute matrix are sorted by the attribute order and the difference between rows is relatively small. *Factoriser* iterates each row and only returns the difference between rows. To build row iterator, we first build the set *end* for each attribute to determine when iterator should propagate the change. For attribute A_i, let itr_{A_i} be the iterator of attribute value in ascending order. The intuition behind the set *end* is that, when itr_{A_i} iterates over any value in set *end*, $\text{itr}_{A_{i+1}}$ should also increment. For example, in Figure 3a, city 2 and city 3 are in *end* because when city iterator itr_C iterates over them, state iterator itr_S should also increment.

Algorithm 1 implements the iterator of attribute rows. Notice that, instead of returning the row values of attribute matrix, it returns the difference between current row and previous row.

Algorithm 1: Row iterator next(A_i, &update) algorithm

```
Result: Update to the previous row for attributes from $A_i$
$\text{itr}_{A_i}$ := current iterator for attribute $A_i$;
nextValue := $\text{itr}_{A_i}$.next();
update[$A_i$] := nextValue;
if current attribute value $\in$ end and $A_i \neq$ root then
    next(parent($A_i$), update);
if !$\text{itr}_{A_i}$.hasNext() then
    $\text{itr}_{A_i}$ := new itr();
return update;
```

D EXPECTATION MAXIMIZATION ALGORITHM

We write the multilevel-model in matrix form where $\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\beta}, \boldsymbol{b}$, and $\boldsymbol{\varepsilon}$ are vertical concatenations of their row-wise vectors/matrices, and $\boldsymbol{Z}$ is a diagonal matrix with $\boldsymbol{X}_i$ along the diagonal:

$$\boldsymbol{y} = \boldsymbol{X} \cdot \boldsymbol{\beta} + \boldsymbol{Z} \cdot \mathbf{b} + \boldsymbol{\varepsilon} \tag{7}$$

EM iterates between two steps. The expectation step uses the estimates $\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\Sigma}}, \hat{\sigma^2}$ to find the expected value of $\hat{\mathbf{b}}_i$, and $\hat{\mathbf{b}}_i \cdot \hat{\mathbf{b}}_i^{\mathrm{T}}$:

$$\boldsymbol{V}_i = \left(\frac{\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i}{\hat{\sigma^2}} + \hat{\boldsymbol{\Sigma}}^{-1}\right)^{-1} \tag{8}$$

$$\boldsymbol{\mu}_i = \frac{\boldsymbol{V}_i \cdot \boldsymbol{X}_i^{\mathrm{T}} \cdot (\boldsymbol{y}_i - \boldsymbol{X}_i \cdot \hat{\boldsymbol{\beta}})}{\hat{\sigma^2}} \tag{9}$$

$$\hat{\mathbf{b}}_i = \boldsymbol{\mu}_i \tag{10}$$

$$\hat{\mathbf{b}}_i \cdot \hat{\mathbf{b}}_i^{\mathrm{T}} = \boldsymbol{V}_i + \boldsymbol{\mu}_i \cdot \boldsymbol{\mu}_i^{\mathrm{T}} \tag{11}$$

The maximization step uses the current estimate of $\hat{\mathbf{b}}$ to estimate the $\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\Sigma}}, \hat{\sigma^2}$ with maximum likelihood:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^{\mathrm{T}} \cdot \boldsymbol{X})^{-1} \cdot \boldsymbol{X}^{\mathrm{T}} \cdot (\boldsymbol{y} - \boldsymbol{Z} \cdot \hat{\mathbf{b}}) \tag{12}$$

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{\mathcal{G}} \cdot \sum_{i=1}^{\mathcal{G}} \hat{\mathbf{b}}_i \cdot \hat{\mathbf{b}}_i^{\mathrm{T}} \tag{13}$$

$$\begin{aligned}\hat{\sigma^2} = \frac{1}{n}((\boldsymbol{y} - \boldsymbol{X} \cdot \hat{\boldsymbol{\beta}})^{\mathrm{T}} \cdot (\boldsymbol{y} - \boldsymbol{X} \cdot \hat{\boldsymbol{\beta}}) + \sum_{i=1}^{\mathcal{G}} \boldsymbol{Tr}(\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i \cdot \mathbf{b}_i \cdot \mathbf{b}_i^{\mathrm{T}}) \\ - 2 \cdot (\boldsymbol{y} - \boldsymbol{X} \cdot \hat{\boldsymbol{\beta}})^{\mathrm{T}} \cdot (\boldsymbol{Z} \cdot \hat{\mathbf{b}}))\end{aligned} \tag{14}$$

where $\boldsymbol{Tr}(\cdot)$ is the trace (sum of main diagonal elements) of a matrix.

Vertical Concatenation: Notice that $\boldsymbol{Z}$ has shape $n \times m \cdot \mathcal{G}$ where $\mathcal{G}$ is the number of clusters (typically exponential in the depth of the attribute in its hierarchy). $\boldsymbol{Z}$ is non-zero along the diagonal, thus its sparsity can be exploited by computing $\boldsymbol{Z} \cdot \hat{\mathbf{b}}$ with vertical concatenation without fully materializing $\boldsymbol{Z}$:

$$\boldsymbol{Z} \cdot \hat{\mathbf{b}} = \text{vertcat}(\boldsymbol{X}_1 \cdot \hat{\mathbf{b}}_1, \boldsymbol{X}_2 \cdot \hat{\mathbf{b}}_2, \ldots, \boldsymbol{X}_{\mathcal{G}} \cdot \hat{\mathbf{b}}_{\mathcal{G}})$$

Multiplication Order: Associative law of matrix multiplication can be exploited to avoid large intermediate result. For example, in equation 12, if matrix chain multiplications are from left to right, there will be an intermediate result with shape $\text{m} \times \text{n}$:

$$\hat{\boldsymbol{\beta}} = \underbrace{((\underset{\text{m}\times\text{n}}{\boldsymbol{X}^{\mathrm{T}}} \cdot \underset{\text{n}\times\text{m}}{\boldsymbol{X}})^{-1} \cdot \underset{\text{m}\times\text{n}}{\boldsymbol{X}^{\mathrm{T}}})}_{\text{m}\times\text{n}} \cdot \underbrace{(\underset{\text{n}\times 1}{\boldsymbol{y}} - \underset{\text{n}\times\text{m}\mathcal{G}}{\boldsymbol{Z}} \cdot \underset{\text{m}\mathcal{G}\times 1}{\hat{\mathbf{b}}})}_{\text{n}\times 1}$$

This could be avoided by reordering matrix multiplications:

$$\hat{\boldsymbol{\beta}} = \underbrace{(\underset{\text{m}\times\text{n}}{\boldsymbol{X}^{\mathrm{T}}} \cdot \underset{\text{n}\times\text{m}}{\boldsymbol{X}})^{-1}}_{\text{m}\times\text{m}} \cdot \underbrace{(\underset{\text{m}\times\text{n}}{\boldsymbol{X}^{\mathrm{T}}} \cdot (\underset{\text{n}\times 1}{\boldsymbol{y}} - \underset{\text{n}\times\text{m}\mathcal{G}}{\boldsymbol{Z}} \cdot \underset{\text{m}\mathcal{G}\times 1}{\hat{\mathbf{b}}}))}_{\text{m}\times 1}$$

Bottleneck: The EM updates above are primarily bottlenecked by six types of matrix multiplication operations: $\boldsymbol{X}^{\mathrm{T}} \cdot \boldsymbol{X}$, $\boldsymbol{X} \cdot \boldsymbol{A}$, $\boldsymbol{B} \cdot \boldsymbol{X}$, $\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i$, $\boldsymbol{X}_i \cdot \boldsymbol{C}_i$, $\boldsymbol{D}_i \cdot \boldsymbol{X}_i$ for $i = 1, ..., \mathcal{G}$, where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}_i, \boldsymbol{D}_i$ are intermediate matrices and $\mathcal{G}$ is the number of clusters. We can precompute $\boldsymbol{X}^{\mathrm{T}} \cdot \boldsymbol{X}$ and $\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i$. We need to perform each other operation once during each iteration.

All of these operations involve $\boldsymbol{X}$, which is the factorised feature matrix. A naive approach is to materialize the full $\boldsymbol{X}$ matrix and use existing matrix operator implementations, but the matrix can be very large. Instead, we wish to directly perform matrix operations on the f-representation.

E MATRIX OPERATIONS

In this section, we provide formal algorithms to compute matrix operations through aggregation queries. We assume that the total number of rows in the relations of each hierarchy is $O(w)$, the number of attributes is d, the number of columns in feature matrix is m and the number rows in feature matrix is n. For simplicity, we assume that feature matrix is the same as attribute matrix. The extension to customized feature matrix is trivial by mapping attribute value to feature value during operations.

Gram Matrix: First consider gram matrix $\boldsymbol{X}^{\mathrm{T}} \cdot \boldsymbol{X}$. The naive multiplication $\boldsymbol{X}^{\mathrm{T}} \cdot \boldsymbol{X}$ has time complexity $O(n \cdot m^2)$. In Figure 13c, the columns in attribute matrix have a lot of redundancy, and, given two columns, we can iterate all attribute values and leverage *COF* to derive how many times two attribute values are duplicated. Note that gram matrix is symmetrical, so we only need to calculate half of the matrix. Let c_i be the ith column and r_i be the ith row of attribute matrix.

Algorithm 2: Gram matrix algorithm

```
Result: $c_i \cdot c_j$
$A_p$ := attribute of $c_i$;
$A_q$ := attribute of $c_j$;
if $A_p == A_q$ then
    return $\frac{TOTAL_{A_d}}{TOTAL_{A_p}} \cdot \sum_{a_p \in Dom(A_p)} COUNT_{A_p}[a_p] \cdot a_p \cdot a_p$ ;
else
    return $\frac{TOTAL_{A_d}}{TOTAL_{A_p}} \cdot \sum_{a_p \in Dom(A_p), a_q \in Dom(A_q)} COF_{A_p,A_q}[a_p, a_q] \cdot a_p \cdot a_q$;
```

Algorithm 2 is used to compute each element of gram matrix $c_i \cdot c_j$ where $i \leq j$. The time complexity to compute each element is $O(w^2)$ and the whole gram matrix is $O(m^2 \cdot w^2)$. Even if attribute matrix has height n exponential in the number of attributes, we can use algorithm 2 to compute gram matrix in time polynomial in m.

Left Multiplication: Next consider left multiplication $\boldsymbol{A} \cdot \boldsymbol{X}$, where the shape of $\boldsymbol{A}$ is $q \times n$. The naive matrix multiplication $\boldsymbol{A} \cdot \boldsymbol{X}$ has time complexity $O(q \cdot n \cdot m)$. Similar to gram matrix, we exploit the fact that the columns in attribute matrix has a lot of redundancy. For each column, we leverage *COUNT* to infer the times each attribute value is duplicated. For ith row r'_i in $\boldsymbol{A}$, we precompute the prefix sum of r'_i in $O(n)$ to get range sum of r'_i in $O(1)$.

Algorithm 3: Left multiplication algorithm

```
Result: $r'_i \cdot c_j$
result := 0;
start := 0;
$A_p$ := attribute of $c_j$;
for  k:= 0; k < $\frac{TOTAL_{A_d}}{TOTAL_{A_p}}$; k++: do
    for  $a_p \in Dom(A_p)$ in ascending order do
        rangeSum := sum($r'_i$[start : start + $COUNT_{A_p}[a_p]$]);
        result+= rangeSum $\cdot a_p$;
        start+= $COUNT_{A_p}[a_p]$;
return result;
```

Algorithm 3 is used to compute each element of left multiplication $c_i \cdot c_j$. Note that the input size is $O(q \cdot n)$ so that the lower bound of the time complexity of algorithm 3 is $O(q \cdot n)$. For each r'_i, the first attribute only needs to iterate over attribute values and compute multiplication result in $O(w)$, while the last attribute can't utilize the prefix sum and have to iterate r'_i in $O(w^m)$. The total time complexity of algorithm 3 is $O(q \cdot (n + w + w^2 + ... + w^m)) = O(q \cdot n)$, which is optimal.

Right Multiplication: Then consider right multiplication $\boldsymbol{X} \cdot \boldsymbol{A}$, where the shape of $\boldsymbol{A}$ is $n \times p$. The naive matrix multiplication $\boldsymbol{X} \cdot \boldsymbol{A}$ has time complexity $O(p \cdot n \cdot m)$. Algorithm 4 uses the row iterator in *Factoriser* to implement right multiplication by updating multiplication result from previous row. Similar to left multiplication, the output size is $O(p \cdot n)$ so that the lower bound of the time complexity of algorithm 4 is $O(p \cdot n)$. For each row iterator, the first attribute is updated $O(w)$ times, while the last attribute is updated $O(w^m)$ times. The total time complexity of algorithm 4 is $O(p \cdot (n + w + w^2 + ... + w^m)) = O(p \cdot w^m) = O(p \cdot n)$, which is optimal.

Algorithm 4: Right multiplication algorithm

```
Result: $r_1 \cdot c'_j, r_2 \cdot c'_j, \cdots, r_n \cdot c'_j$
$r_{prev}$ := the first values for all attributes;
$r_1 \cdot c'_j = r_{prev} \cdot c'_j$;
for  k:= 2; k <= n ; k++: do
    A := last attribute in attribute order;
    update := new map();
    update = RowItr.next(A, update);
    $r_n \cdot c'_j = r_{n-1} \cdot c'_j$ ;
    for attribute $A_i$, value v ∈ update do
        $r_n \cdot c'_j$ −= $r_{prev}[i] \cdot c'_j[i]$;
        $r_n \cdot c'_j$ += $v \cdot c'_j[i]$;
        $r_{prev}[i] = v$;
```

F MATRIX OPERATIONS OVER CLUSTERS

We study the matrix operations over each cluster of attribute matrix ($\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i$, $\boldsymbol{X}_i \cdot \boldsymbol{C}_i$, $\boldsymbol{D}_i \cdot \boldsymbol{X}_i$ where $i = 1, ..., \mathcal{G}$) in this section. Given the initial view $V = \gamma_{A_{gb}, F_{agg}(A_{agg})}(\mathbb{R})$, we call A_{gb} inter cluster attributes. After user drill-down to a hierarchy, the additional attribute S in A'_{gb} is called intra cluster attribute. Because we previously require that intra cluster attribute is placed last in the attribute order, the rows in the same cluster are adjacent, so we can reuse the row iterator to iterate through clusters. We exploit the fact that, for each cluster, inter cluster attributes have the same value and reuse the row iterator to only calculate the difference between clusters. We update the previous matrix according to the difference. We also assume that attribute matrix is the same as attribute matrix for simplicity.

Gram Matrices: First consider gram matrices for all clusters $\boldsymbol{X}_i^{\mathrm{T}} \cdot \boldsymbol{X}_i$ for $i = 1, ..., \mathcal{G}$. The naive implementation takes $O(m^2 \cdot w \cdot \mathcal{G}) = O(n \cdot m^2)$. Algorithm 5 computes the gram matrix for each cluster by iterating over each cluster and updating the difference. Notice that, the updates are in place and the outputs are read-only except for the last output. Even if we reuse the same matrix, the matrix is yielded $\mathcal{G}$ times and each time at least $O(m)$ elements need to be changed, so the lower bound of time complexity is $O(m \cdot \mathcal{G})$. The first inter cluster attribute is updated $O(w)$ and the last is updated $O(w^{m-1})$. Each update involve $O(m)$ changes in the matrix. Change for intra attributes takes $O(m)$ for $O(\mathcal{G})$ times. Therefore, the total time complexity is $O(m \cdot (w + ... + w^{m-1} + \mathcal{G})) = O(m \cdot \mathcal{G})$ which is optimal.

Algorithm 5: Cluster gram matrix iterator algorithm

```
Result: Gram matrix for each cluster
$r_{inter}$ := values of inter cluster attributes in the first cluster;
$r_{intra}$ := value sums of intra cluster attributes in the first cluster;
gram := compute gram matrix for the first cluster naively;
prevSize := number of tuples in the first cluster;
yield gram;
for k:= 2; k <= n ; k++: do
    A := last attribute among inter cluster attributes;
    update := new map();
    update = RowItr.next(A, update);
    curSize := number of tuples in kth cluster;
    for attribute $A_i$, value v ∈ update do
        for j:= 1; j <= number of inter cluster attributes; j++: do
            gram[i, j] /= $r_{inter}$[i];
            gram[i, j] *= v;
            gram[i, j] *= curSize/prevSize;
        $r_{inter}$[i] = v;
    /* Cache given intra cluster attribute value */
    for each pair of intra cluster attributes do
        Update corresponding gram matrix elements naively;
    for attribute $a_i$ ∈ intra cluster attributes do
        sum := sums of values of attribute $a_i$ in kth cluster;
        for j:= 1; j <= number of inter cluster attributes; j++: do
            gram[i, j] /= $r_{intra}$[i];
            gram[i, j] *= sum;
        $r_{intra}$[i] = sum;
    prevSize := curSize;
    yield gram;
```

Left Multiplication: Next consider left multiplication for all clusters $\boldsymbol{A}_i \cdot \boldsymbol{X}_i$ for $i = 1, ..., \mathcal{G}$, where the shape of $\boldsymbol{A}_i$ is $q \times n_i$. The naive implementation takes $O(q \cdot n \cdot m)$. Algorithm 6 computes the left multiplication for each cluster by iterating over each cluster and updating the difference. The input size is $O(q \cdot n)$ so that the lower bound of the time complexity is $O(q \cdot n)$. Each row in each $\boldsymbol{A}_i$ takes $O(m + w)$. Therefore, the total time complexity is $O(q \cdot \mathcal{G} \cdot (m + w)) = O(q \cdot m \cdot \mathcal{G} + q \cdot n)$.

Algorithm 6: Cluster left multiplication iterator algorithm

```
Result: Left multiplication with $r'_{i,k}$ for kth cluster
$r_{inter}$ := values of inter cluster attributes in the first cluster;
result := compute result for the first cluster naively;
yield result;
for k:= 2; k <= n ; k++: do
    A := last attribute in inter cluster attributes;
    update := new map();
    update = RowItr.next(A, update);
    rowSum := sum($r'_{i,k}$);
    for attribute $A_i$, value v ∈ update do
        $r_{inter}$[i] = v;
    for each inter cluster attribute $A_i$ do
        result[i] = $r_{inter}$[i] * rowSum;
    for attribute A ∈ intra cluster attributes do
        Update corresponding result matrix elements naively;
    yield result;
```

Right Multiplication: Finally, consider right multiplication for all clusters $\boldsymbol{X}_i \cdot \boldsymbol{A}_i$ for $i = 1, ..., \mathcal{G}$, where the shape of $\boldsymbol{A}_i$ is $m \times p$. The naive implementation takes $O(p \cdot n \cdot m)$. Algorithm 7 computes the right multiplication for each cluster. The output size is $O(p \cdot n)$ and, for each cluster, we can always find $\boldsymbol{A}_i$ such that all elements in the output have to change. Therefore the lower bound of the time complexity is $O(p \cdot n)$. Each column in each $\boldsymbol{A}_i$ takes $O(m + w)$. Therefore, the total time complexity is $O(p \cdot \mathcal{G} \cdot (m + w)) = O(p \cdot m \cdot \mathcal{G} + p \cdot n)$.

Algorithm 7: Cluster right multiplication iterator algorithm

```
Result: Right multiplication with $c'_{i,k}$ for kth cluster
$r_{inter}$ := values of inter cluster attributes in the first cluster;
result := compute result for the first cluster naively;
yield result;
for k:= 2; k <= n ; k++: do
    A := last attribute in inter cluster attributes;
    update := new map();
    update = RowItr.next(A, update);
    for attribute $A_i$, value v ∈ update do
        $r_{inter}$[i] = v;
    base = $\sum_{j=0}^{\text{number of inter cluster attributes}} r_{inter}[j] \times c'_{i,k}[j]$;  for
    j:= 1; j <= $n_k$; j++: do
        value := 0;
        for attribute $A_i$ ∈ intra cluster attributes do
            value += jth value of $A_i$ in kth cluster × $c'_{i,k}$[i]
        result[j] := base + value;
    yield result;
```

Evaluation: We evaluate the performance of matrix operation using synthetic datasets with d hierarchies. For each hierarchy, there are three attributes. Each attribute contains $w = 10$ unqiue values. Given d attributes, the total number of rows $n = 10^d$. $\boldsymbol{X}$ has the shape $10^d \times 3 \cdot d$ and each cluster $\boldsymbol{X}_i$ has the shape $10 \times 3 \cdot d$ for $i = 1, ..., \mathcal{G}$. There are 10^{d-1} clusters in total. For right Multiplications over clusters $\boldsymbol{X}_i \cdot \boldsymbol{C}_i$, $\boldsymbol{C}_i$ has the shape $3 \cdot d \times 1$. For

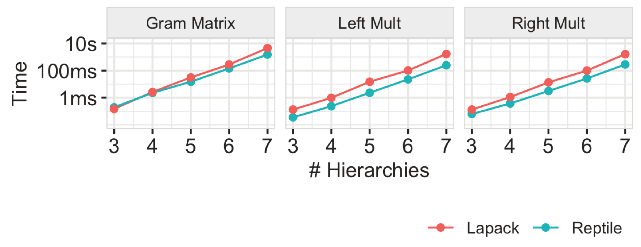


Figure 14: Matrix operation over clusters runtimes compared to Lapack-based implementation.

each Left Multiplications over clusters $\boldsymbol{D}_i \cdot \boldsymbol{X}_i$, $\boldsymbol{D}_i$ has the shape 1×10. We randomly generate matrix to be multiplied.

Figure 14 reports runtimes in log scale. At 7 hierarchies, `Reptile` is 3× faster for gram matrix, 5.8× faster for left multiplication, and 6.9× faster for right multiplication. Overall, `Reptile` outperforms Lapack.

G MATRIX OPERATION OVER GENERAL FACTORISED REPRESENTATION

In this section, we briefly discuss how to extend Matrix operations over general factorised representation.

Given general F-tree, we first need to determine attribute order. We require that for any pair of attributes in attribute order, attribute before doesn't transitively depends on attribute after. This requirement is to ensure that row iterator can work properly. Iterator for attribute after should increment first and propagate the change to iterator for attribute before.

The first extension is, for relation without functional dependency, the join operator and aggregation operator need to record the order of tuples even if tuples have same value. Consider the following example:

Example 9 (Order in operator). *Given relation* $\mathrm{R} = [(\mathrm{a}_1, \mathrm{b}_1), (\mathrm{a}_1, \mathrm{b}_2), (\mathrm{a}_2, \mathrm{b}_1)]$ *over schema* $\mathrm{S} = [\mathrm{A}, \mathrm{B}]$*, where there is no functional dependency. After marginalize out attribute A, the result is an ordered map* $\bigoplus_{\mathrm{A}} \mathrm{R} = \{\mathrm{b}_1 : 2, \mathrm{b}_2 : 1\}$*. However, the information that* b_2 *is in middle of two* b_1 *is lost, which is necessary during multiplication as we need to infer the positions. One solution would be that, aggregation operator returns an ordered list:* $\bigoplus_{\mathrm{A}} \mathrm{R} = [\mathrm{b}_1 : 1, \mathrm{b}_2 : 1, \mathrm{b}_1 : 1]$.

The second extension is to redefine the aggregation query. Given a set of attributes S, let *dep*(S) be the dependency set of S. Dependency set *dep*(S) is the set with minimum size that, for each attribute in *dep*(S), its dependency is also in *dep*(S). Let *rel*(S) be the set of relations whose schema contains any attribute in S. Let *after*(A) be the set of attributes after A in attribute order including A.

Then aggregation queries are redefined as:

$$TOTAL_{\mathrm{A_k}} = \bigoplus_{dep(after(\mathrm{A}))} \bigotimes_{rel(dep(after(\mathrm{A})))} \mathrm{R}$$

$$COUNT_{\mathrm{A_k}} = \bigoplus_{dep(after(\mathrm{A}))/\{\mathrm{A_k}\}} \bigotimes_{rel(dep(after(\mathrm{A})))} \mathrm{R}$$

$$COF_{\mathrm{A_k},\mathrm{A_j}} = \bigoplus_{dep(after(\mathrm{A}))/\{\mathrm{A_k},\mathrm{A_j}\}} \bigotimes_{rel(dep(after(\mathrm{A})))} \mathrm{R}$$

In general, the dependency set *dep*(*after*(A))) may include all attributes if all attributes have no-empty dependency. Because operator needs to store the order of attribute value, in the worst case, the join result may be as large as the fully joined relation even if attributes are marginalized early.

H MULTI-ATTRIBUTE FEATURES

In this section, we discuss the extension to multi-attribute features. If the number of attributes is a constant, previous time complexity analyses still apply. In the worst case, if all the features are multi-attribute features related to all the attributes, there would be no redundancy in feature matrix, and our solution would be the same as the naive solution.

For multi-attribute external feature, user may have dataset that maps multiple attributes to feature values. Given a list of attribute $\mathbb{A}$ with k attributes, tuple of attribute value $(a_1, ..., a_k) \in \mathrm{Dom}(\mathbb{A})$ and aggregation function $\mathrm{F_{agg}}$, assume that there is an external dataset $\mathrm{R_{external}}$ which maps $(a_1, ..., a_k)$ to its feature value, the external feature is then:

$$\mathrm{feature_{external}}[(a_1, ..., a_k)] = \mathrm{R_{external}}[(a_1, ..., a_k)]$$

To register multi-attribute external feature, user needs to provide external dataset $\mathrm{R_{external}}$, a list of attributes $\mathbb{A}$ and target aggregation function $\mathrm{F_{agg}}$.

For multi-attribute custom feature, given a list of attribute $\mathbb{A}$ with k, tuple of attribute value $(a_1, ..., a_k) \in \mathrm{Dom}(\mathbb{A})$ and aggregation function $\mathrm{F_{agg}}$, the custom feature is then:

$$\mathrm{feature}[(a_1, ..., a_k)] = \sigma_{pred}(\gamma_{\mathbb{A},F'}(Q))$$

where *pred* is the predicate of selection, and F' is the aggregation function which user can customize. User needs to provide the predicate *pred*, aggregation function F', a list of attributes $\mathbb{A}$ and the target aggregation function $\mathrm{F_{agg}}$ to register derived variable as feature.

For multi-attribute feature registered with a list of attributes $\mathbb{A}$ and target aggregation function F, given the view $\mathrm{V}' = \gamma_{\mathrm{A}'_{gb},\mathrm{F_{agg}}(\mathrm{A_{agg}})}(\mathrm{R})$, this feature is applicable if $\mathbb{A} \subseteq \mathrm{A}'_{gb} \wedge \mathrm{F_{agg}} = F$.

For feature matrix, all the multi-attribute features are appended to end of columns. Hierarchy order, attribute order and attribute matrix remain unchanged.

Algorithm 8: gram matrix algorithm for multi-attribute features

```
Result: $c_i \cdot c_j$
$f_i$ := feature for $c_i$;
$f_j$ := feature for $c_j$;
$\mathbb{A}_p$ := list of attribute of $f_i$;
$\mathbb{A}_q$ := list of attribute of $f_j$;
$\mathbb{A} := \mathbb{A}_q \cup \mathbb{A}_p$;
$k$ := size of $\mathbb{A}$ ;
$A_{first}$ := first attribute in $\mathbb{A}$;
$A_{last}$ := last attribute in $\mathbb{A}$;
$\mathbb{A}_{all}$ := all attributes before $A_{last}$ ;
$COF = \bigoplus_{\mathbb{A}_{all}/\mathbb{A}} \pi_{A_{last}}(R_{last}) \bigotimes_{i\in[_{last}-1]} R_i$ ;
return $\frac{TOTAL_{A_d}}{TOTAL_{A_{first}}} \cdot \sum_{(a_1,...,a_k)\in \mathrm{Dom}(\mathbb{A})} COF\,[(a_1,...,a_k)] \cdot f_i(\sigma_{\mathbb{A}_p}(a_1,...,a_k)) \cdot f_j(\sigma_{\mathbb{A}_q}(a_1,...,a_k))$ ;
```

For matrix operations, first consider gram matrix. Algorithm 8 computes gram matrix element and Algorithm 9 computes left multiplication for multi-attribute features. We assume that attributes in $\mathbb{A}_p$, $\mathbb{A}_q$ and $\mathbb{A}$ are ordered by the attribute order. Right multiplication is similar to algorithm 4, except that, for each update, the change is $f_{idx}((a_1,...,a_k)) \cdot c'_j[idx]$ instead of $f_{idx}(a) \cdot c'_j[idx]$.

Algorithm 9: Left multiplication algorithm for multi-attribute features

```
Result: $r'_i \cdot c_j$
result := 0;
start := 0;
$f_j$ := feature of $c_j$;
$\mathbb{A}_p$ := list of attribute of $f_j$;
$k$ := size of $\mathbb{A}_p$ ;
$A_{first}$ := first attribute in $\mathbb{A}_p$;
$A_{last}$ := last attribute in $\mathbb{A}_p$;
$\mathbb{A}_{all}$ := all attributes before $A_{last}$ ;
$COF = \bigoplus_{\mathbb{A}_{all}/\mathbb{A}_p} \pi_{A_{last}}(R_{last}) \bigotimes_{i\in[_{last}-1]} R_i$ ;
for $k$:= 0; $k < \frac{TOTAL_{A_d}}{TOTAL_{A_{first}}}$; $k$++: do
    for $(a_1,...,a_k) \in \mathrm{Dom}(\mathbb{A}_p)$ in ascending order do
        rangeSum := sum($r'_i$[start : start + $COF\,[(a_1,...,a_k)]$]);
        result+= rangeSum $\cdot f_j((a_1,...,a_k))$;
        start+= $COF\,[(a_1,...,a_k)]$;
return result;
```

I MULTI-QUERY EXECUTION

Suppose there are d attributes in attribute order. For each model training, there are $2d + \frac{d(d-1)}{2}$ queries to execute. One naive way to execute these queries is to join all relations together and apply aggregation function.

We can rewrite the queries such that these quries can reuse results from other queries:

$$\begin{aligned}
COUNT_{A_1} &= \pi_{A_1}(R_1) \\
COUNT_{A_k+1} &= \bigoplus_{A_k} COF_{A_{k+1},A_k} \text{ for } k = 2,...,d-1 \\
TOTAL_{A_k} &= \bigoplus_{A_k} COUNT_{A_k} \text{ for } k = 1,...,d \\
COF_{A_k,A_{k-1}} &= \pi_{A_k}(R_k) \bigotimes R_{k-1} \bigotimes COUNT_{A_k} \text{ for } k = 2,...,d \\
COF_{A_k,A_j} &= \bigoplus_{A_{k-1}} \pi_{A_k}(R_k) \bigotimes R_{k-1} \bigotimes COF_{A_{k-1},A_j} \\
&\quad \text{for } k = 1,...,d, j = 1,...,d, k > j+1
\end{aligned}$$

Algorithm 10 leverages the dependency to compute query results. The naive solution materializes the join result and apply aggregation functions with total time complexity $O(d^2 \cdot w^d)$. For algorithm 10, attributes in join results are marginalized as soon as possible when they are no longer used in the future queries. The join results are also stored in factorised representations. For *COF* between different hierarchies, we are computing the Cartesian Products. `Reptile` exploits the independence by storing factorised representation. For implementation, only pointers to two relations are stored in O(1). Because we assume that the total number of rows in the relations of each hierarchy is $O(w)$, join operator between attributes in the same hierarchy takes $O(w)$. Suppose there are $O(|\mathbb{H}|)$ hierarchies, each with $O(t)$ attributes and $O(|\mathbb{H}| \cdot t) = O(d)$. The total time complexity for algorithm 10 is $O(|\mathbb{H}|^2 \cdot t^2 + |\mathbb{H}| \cdot t^2 \cdot w)$. If w is much larger than $|\mathbb{H}|$, the time complexity is then $O(|\mathbb{H}| \cdot t^2 \cdot w)$.

Algorithm 10: Mutiple query plan

```
Result: Query results
$COUNT_{A_1} := \pi_{A_1}(R_1)$;
$TOTAL_{A_1} = \bigoplus_{A_1} COUNT_{A_1}$;
$COF_{A_2,A_1} = \pi_{A_2}(R_2) \bigotimes R_1 \bigotimes COUNT_{A_1}$
for $i$ := 3; $i$ <= d; $i$++ do
    $COF_{A_i,A_1} = \bigoplus_{A_{i-1}} \pi_{A_i}(R_i) \bigotimes R_{i-1} \bigotimes COF_{A_{i-1},A_1}$;
for $i$ := 2; $i$ <= d; $i$++ do
    $COUNT_{A_i} = \bigoplus_{A_{i-1}} COF_{A_i,A_{i-1}}$;
    $TOTAL_{A_i} = \bigoplus_{A_i} COUNT_{A_i}$;
    if $i$ < d then
        $COF_{A_{i+1},A_i} = \pi_{A_{i+1}}(R_{i+1}) \bigotimes R_i \bigotimes COUNT_{A_i}$;
    for $j$ := $i$+2; $j$ <= d; $j$++ do
        $COF_{A_j,A_i} = \bigoplus_{A_{j-1}} \pi_{A_j}(R_j) \bigotimes R_{j-1} \bigotimes COF_{A_{j-1},A_i}$;
```

J DRILL-DOWN

Drilling down an attribute involves two steps:

1. append the attribute to the corresponding hierarchy.
2. move the hierarchy to the end of hierarchy order.

After the drill-down operation, F-tree has an additional attribute, and all attributes in one hierarchy is moved to the bottom of the tree. One naive way to implement drill-down operation is to rebuilt the F-tree and recompute all aggregation results from scratch. Assume that w is much larger than $|\mathbb{H}|$, the time complexity to drill-down all hierarchies is then $O(|\mathbb{H}|^2 \cdot t^2 \cdot w)$.

We then introduce optimization to reuse the aggregation results from the previous drill-down. The main property we exploit is the independence between hierarchies. Given the aggregation query over the Cartesian's Product, we can marginalize each relation before join:

$$\bigoplus_{A \in S} \bigotimes_{i \in [k]} R_i[S_i] = \bigotimes_{i \in [k]} (\bigoplus_{A \in S_i} R_i[S_i])$$

where k is the number of relations, S_i is the schema of R_i and $S = \bigcup_{i \in [k]} S_i$ is the schma of the join result. For $i \neq j$, $S_i \cap S_j = \emptyset$.

Notice that, between different hierarchies, we need to compute cartesian product. Suppose that there are t hierarchies. For each hierarchy D_i for $i = 1, ..., t$, let $[D_i]$ be the set of indices of attributes under this hierarchy. Given hierarchy order $D_t, ..., D_1$, define:

$$TOTAL_{D_k} = \bigoplus_{A_i : i \in [D_k]} \bigotimes_{i \in [D_k]} R_i$$

for $k = 1, ..., t$. $TOTAL_{D_k}$ outputs the number of tuples in the hierarchy D_k.

Therefore, we can rewrite all the queries to exploit the independence between hierarchies. Assume that attribute A_k is in hierarchy D_s, attribute A_j is in D_v and $k > j$:

$$\begin{aligned}
&TOTAL_{A_k} \\
&= \bigoplus_{A_1} \cdots \bigoplus_{A_k} \pi_{A_k}(R_k) \bigotimes_{i \in [k-1]} R_i \\
&= (\bigoplus_{A_i : i \in [D_d]} \bigotimes_{i \in [D_d]} R_i) \bigotimes \cdots (\bigoplus_{A_i : i \in [D_{s-1}]} \bigotimes_{i \in [D_{s-1}]} R_i) \bigotimes \\
&\quad (\bigoplus_{A_i : i \in [D_s] \wedge i \leq k} \pi_{A_k}(R_k) \bigotimes_{i \in [D_s] \wedge i < k} R_i) \\
&= \bigotimes_{i \in [s-1]} TOTAL_{D_i} \bigotimes (\bigoplus_{A_i : i \in [D_s] \wedge i \leq k} \pi_{A_k}(R_k) \bigotimes_{i \in [D_s] \wedge i < k} R_i) \\
&\quad \text{for } k = 1, ..., d
\end{aligned}$$

$$\begin{aligned}
&COUNT_{A_k} \\
&= \bigotimes_{i \in [s-1]} TOTAL_{D_i} \bigotimes (\bigoplus_{A_i : i \in [D_s] \wedge i < k} \pi_{A_k}(R_k) \bigotimes_{i \in [D_s] \wedge i < k} R_i) \\
&\quad \text{for } k = 1, ..., d
\end{aligned}$$

$$\begin{aligned}
&COF_{A_k, A_j} \\
&= \bigotimes_{i \in [v-1]} TOTAL_{D_i} \bigotimes (\bigoplus_{A_i : i \in [D_v] \wedge i < j} \pi_{A_j}(R_j) \bigotimes_{i \in [D_v] \wedge i < j} R_i) \\
&\quad \bigotimes_{i \in [v+1, s-1]} TOTAL_{D_i} \bigotimes (\bigoplus_{A_i : i \in [D_s] \wedge i < k} \pi_{A_k}(R_k) \bigotimes_{i \in [D_s] \wedge i < k} R_i) \\
&\quad \text{for } k = 1, ..., d, j = 1, ..., d, k > j
\end{aligned}$$

After rewriting the queries, we can exploit the fact that, when drill-down attribute A_k is in hierarchy D_s, for $TOTAL_{A_i}$, $COUNT_{A_i}$ and COF_{A_i, A_j} where A_i and A_j are not in hierarchy D_s, only the parts $\bigotimes TOTAL_D$ are affected, which are scalars. For join operator $\bigotimes$, when multiplied by scalar, we don't need to apply the multiplication to each tuple. We can maintain a scalar for each relation as the zoom value so that multiplication by scalar is in O(1).

Algorithm 11 shows how to update the aggregation results after drill-down. For aggregation results involved with only the attributes in the hierarchy to drill-down, we have to recompute them in $O(t^2 \cdot w)$. However, for other attributes, the updates can be done in O(1). The total time complexity would be $O(t^2 \cdot w)$.

In algorithm 11, we also cache $TOTAL'$, $COUNT'$, COF', and $TOTAL'_{D_v}$ involved with only attributes in the hierarchy to drill-down because, given the query in equation 2 and drill-down hierarchy H, these aggregation results will always be the same independent of the current view. Consider a scenario when users make complaint twice. For the first complaint, `Reptile` drills down each hierarchy in $\mathbb{H}$ and selects one optimal hierarchy $H^* \in \mathbb{H}$ as in Equation (1). The time complexity for the first complaint is $O(|\mathbb{H}| \cdot t^2 \cdot w)$. For the second complaint, without cache, `Reptile` needs to recompute each hierarchy in $\mathbb{H}$ and the total time is also $O(|\mathbb{H}| \cdot t^2 \cdot w)$. If all the attributes not selected in the first complaint $\mathbb{H}/\{H^*\}$ are cached, each cached hierarchy can be updated in $O(d^2)$ for the second complaint. So the total time for the second complaint would be $O(t^2 \cdot w)$.

Algorithm 11: drill-down hierarchy D_v

```
Result: Updated query results after drill-down
$A_{new}$:= Attribute in $D_v$ to drill-down;
$A_u, \cdots, A_{u+t}$ := Attributes in $D_v$ from lowest to highest level;
/* Cache TOTAL', COUNT', COF', and $TOTAL'_{D_v}$ */
Compute updated $TOTAL'$, $COUNT'$ and $COF'$ involved with
only attributes $A_{new}, A_u, \cdots, A_{u+t}$ using Algorithm 10;
$TOTAL'_{D_v} = \bigoplus_{A_i : i \in [D_v]} \bigotimes_{i \in [D_v]} R_i$;
for Attribute $A_k \notin \{A_{new}, A_u, \cdots, A_{u+t}\}$ do
    if k < u then
        $COUNT'_{A_k} = COUNT_{A_k} \bigotimes TOTAL'_{D_v}$;
        $TOTAL'_{A_k} = TOTAL_{A_k} \bigotimes TOTAL'_{D_v}$;
    else
        $COUNT'_{A_k} = COUNT_{A_k} \bigotimes (TOTAL'_{D_v} / TOTAL_{D_v})$;
        $TOTAL'_{A_k} = TOTAL_{A_k} \bigotimes (TOTAL'_{D_v} / TOTAL_{D_v})$;
    for Attribute $A_j \in \{A_{new}, A_u, \cdots, A_{u+t}\}$ do
        $COF_{A_k, A_j} = COUNT'_{A_k} \bigotimes COUNT'_{A_j} / TOTAL'_{D_v}$
    for Attribute $A_j \notin \{A_{new}, A_u, \cdots, A_{u+t}\} \wedge k > j$ do
        if u > k > j then
            $COF'_{A_k, A_j} = COF_{A_k, A_j} \bigotimes TOTAL'_{D_v}$
        else
            $COF'_{A_k, A_j} = COF_{A_k, A_j} \bigotimes (TOTAL'_{D_v} / TOTAL_{D_v})$
```

K QUALITY OF MULTI-LEVEL MODEL

We conduct model evaluation between linear regression model and multi-level with default features only, and with external features. The following two datasets are considered:

FIST: This dataset contains the farmer reported drough severity at different villages in different years in Ethiopia. There are 2 hierarchies: year (one attribute with 36 values), location (three attributes: region, district and village, with 161 village values). Sensing data of rainfall are available each year for each village, which are used

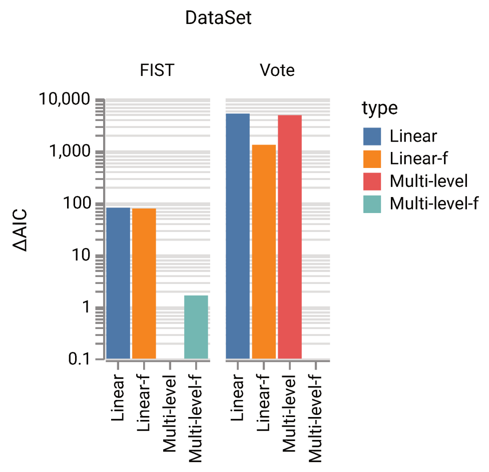


Figure 15: Model evaluation

as external feature. The mean drought severity has been estimated using different models.

Vote: This dataset contains the 2020 presidential election vote results at different counties in the United States. There are 1 hierarchies: location (two attributes: state, and county, with 3147 county values). 2016 presidential election vote results at different counties are available, which are used as external feature. The percentage of votes for Donald Trump has been estimated using different models.

To evaluate model performance, we use Akaike information criterion (AIC) [8], which estimates the qualities of a collections of models. AIC makes the trade-off between both goodness of fit of the model and the simplicity of the model. Given the same set of data, model with low AIC scores are considered to be relatively better. For each dataset, the difference of AIC: $\Delta AIC_i = AIC_i - AIC_{min}$ for ith model is shown, where AIC_{min} is the lowest AIC among the collection of models. As a rule of thumb, for the same dataset, one model is considered to be substantially better than the other if the difference of AIC is larger than 10 [14].

The result of model evaluation is shown in Figure 15. Linear is the linear regression model with only default features. Linear-f is the linear regression model with both default and auxiliary features. Multi-level is the multi-level model with default features only. Multi-level-f is the multi-level model with both default and auxiliary features. For FIST dataset, multi-level models are substantially better than linear regression models. For Vote dataset, multi-level model with auxiliary feature is substantially better than linear regression model with auxiliary feature. Because the vote results in 2016 are strong predictors of vote results in 2020, models with auxiliary feature are substantially better than models without auxiliary feature.

L CASE STUDY DETAILS: COVID-19

TODO: compare against state-of-art error detection. This also shows the limitations of data cleaning: don't work well for discrete data type.

While they are good at categorical values, not good for numerical

While tehre are systems that repair numeric values (Eracer and Data Cleaning Using Belief Propagation) based on modeling, their models are domain specific and are not applicable.

Reptile provides a default for datasets with hiearchy to capture features in different dimensions, which is cmommon, and allows domain-specific auxiliary dataset, which has already been commonly used for data science.

Domain knowledge about COVID that are easy for human to provide but hard for system to infer.

Similar work (CleanML) has also shown that advanced data cleaning method like holoclean is not significantly better than naive methods.

In this section, we discuss the details of COVID-19 case study.

We first discuss the basic setting of `Reptile`. COVID-19 dataset includes global data and United States data. For global data, because of the large number of countries, we further cluster countries by regions. `Reptile` use 1 day and 7 day lag features for trend and seasonality. Given data cleaning issues, we create complaint at the higher level of geographical hierarchy. For instance, given issue that the total confirmed cases in Texas is under reported on Jan 21 2021, we complain that the total confirmed cases is too low in the United States on that day. (It is also possible to make complaint at the higher level of time hierarchy. In the experiment, we only make complaint about one day because, for COVID-19 dataset, people tend to focus on daily number across different locations. For all issues we studied, people make complaint about different locations on one specific day instead of the whole month/year.) We then check if different approaches can successfully recommend the cluster with data cleaning problems.

ID	Issue	RP	ST	SP	H	Ra
3572	Texas confirmed missing reports	✓				
3521	Arizona death methodology altered	✓				
3482	Washington missing reports	✓				
3476	★ Utah missing source					
3468	New York death missing reports	✓				
3466	Montana missing reports	✓				
3456	North Dakota confirmed backlog	✓				
3451	Iowa death missing reports	✓				
3449	Arizona test over reported	✓				
3448	Washington death wrongly reported	✓				
3441	★ Albany confirmed day shift					
3438	Ohio confirmed backlog	✓				
3424	Massachusetts confirmed backlog					
3416	Nevada death over reported	✓				
3414	Eureka death over reported	✓				
3402	Washington confirmed typo					

Table 1: List of COVID-19 issues in the US. RP is `Reptile`, ST `Sensitivity`, and SP is `Support`, H is Holoclean, Ra is Raha. Prevalent errors are highlighted with ★.

The details of issues for US and global are in Table 1 and Table 2 respectively. We highlight one type of errors: prevalent errors. Prevalent errors are defined as errors widespread across all time or locations. For example, some sources of confirmed and death are

ID	Issue	RP	ST	SP	H	Ra
3623	Germany recovered over reported	✓				
3618	★ Quebec death missing source					
3578	US recovery nullified	✓	✓			
3567	India confirmed missing reports	✓				
3546	★ Thailand confirmed missing source					
3538a	Mexico confirmed definition altered	✓				
3538b	Mexico confirmed missing reports	✓				
3518	★ Sweden death missing source					
3498	★ Alberta missing source				✓	
3494	UK death missing reports	✓				
3471	Turkey confirmed definition altered	✓	✓	✓		
3423	Afghanistan confirmed wrongly reported					
3413	France missing reports	✓				
3408	Kazakhstan confirmed over reported	✓				

Table 2: List of global COVID-19 issues.

missing over the course of the pandemic in Utah, which affects data all the time and makes result inconsistent with official report on Dec 18 2020. The other non-prevalent common issues are missing report (e.g. the reports of confirmed cases in Texas are missing on Jan 15 2021), data backlog (e.g. confirmed cases are not fully updated for North Dakota, and spike on Dec 9 2020), change of definition (e.g., Arizona updated guidance for identifying deaths, which causes abnormally high deaths on Jan 5 2021), etc.

Overall, Reptile outperforms Sensitivity and Support because Sensitivity and Support only recommend outliers. For example, given complaint that the COUNT of confirmed cases is too high, Sensitivity and Support always choose the location with the highest COUNT of confirmed cases, disregarding the fact that these locations have the highest population and the high COUNT is normal.

Next, we discuss issues which Reptile fail to identify. Reptile fail to detect all prevalent errors. Since prevalent errors repeat across large number of clusters, Reptile is unable to tell if these clusters are all normal or all problematic. Besides prevalent errors, Reptile is unable to identify errors whose effects are not strong enough and are masked by noises from other clusters. For issue 3424, there is a backlog of 680 confirmed cases in Massachusetts on Dec 18 2020, which is relatively small given that there are 290578 total confirmed cases and 4853 new cases in Massachusetts on that day. For issue 3423 there is a decrease of confirmed case from 46980 to 46718 on Dec 3 2020 which is relatively small. For issue 3402, there is a typo for the number of confirmed cases in Washington on Dec 18 2020, whose difference is relatively small.

M CASE STUDY DETAILS: FIST

In this section, we show the user interface and discuss two complaints that our system fail to identify all causes.

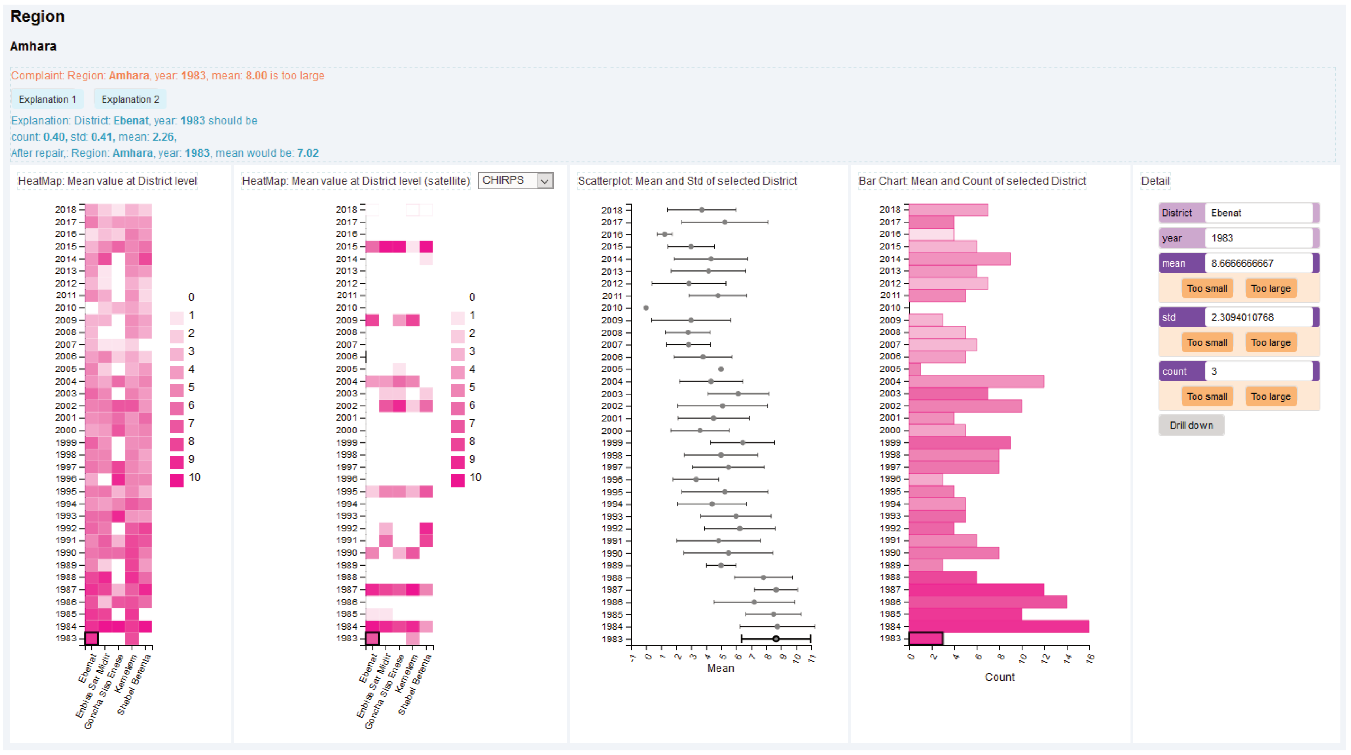
Figure 16 shows the user interface for the study. Here, participant has made a complaint about Region Amhara. At the top, two explanations are generated that highlights two Districts which, if their aggregation results are repaired, can resolve complaint. The first heatmap shows drought severity for Districts in Amhara. The second heatmap shows the remote sensing. The scatterplot and barchart visualize aggregations results (AVG, STD and COUNT). Participant can further make complaint at District level.

For the first complaint, one team member recalls that one year is a severe year and complains that the mean severity of one region is too low. However, it turns out that all the districts in this region have low mean severity. For sensing data, some of them indicate that this year is severe, but some of them don't. Different team members also hold different opinions about this year. More investigations are needed to understand this complaint.

For the second complaint, one team member complains that the standard deviation of one regions is too high. The error is caused by two districts, but our system only identify one district. The failure is because of the property of the standard deviation. When the complaints are caused by multiple clusters, repair only one cluster may not cause the standard deviation closer to the true value. Consider the following minimum example:

Suppose there are three same values n initially. The initial mean is n and variation is 0. Suppose we corrupt first two values by adding Δ: $n+\Delta$, $n+\Delta$, and n. The mean becomes $n+\frac{2}{3}\Delta$ and variation becomes $\frac{2}{3}\Delta^2$. Suppose we fix the first corruption: n, $n+\Delta$, and n. The mean becomes $n+\frac{1}{3}\Delta$ and variation becomes $\frac{2}{3}\Delta^2$. Notice that variation is the same as that before fix. Suppose user complains about the high standard deviation, fixing any of two corrupted values wouldn't resolve user's complaint. Suppose we fix the first corruption partly to Δ': $n+\Delta'$, $n+\Delta$, and n. The mean becomes $n+\frac{1}{3}\Delta+\frac{1}{3}\Delta'$ and variation becomes $\frac{2}{3}(\Delta^2-\Delta\Delta'+\Delta'^2)$. Let variation be a function of Δ': $f(\Delta')=\frac{2}{3}(\Delta^2-\Delta\Delta'+\Delta'^2)$. This is a parabola with turning point $(\frac{1}{2}\Delta, \frac{1}{2}\Delta^2)$. That is, the minimum standard deviation is achieved by fixing half of the corruption.

One solution is that, for equation 5 in the optimization problem, we search for a set tuples $\alpha \subseteq Q$ instead of one tuple. This makes the optimization problem NP-hard because, given n tuples in Q, there are 2^n possible subsets of tuples. Joglekar et al. [36] exploits the property of submodularity to greedily search the optimal solution, while in our case, submodularity can't be guaranteed. Another solution is to relax the boolean constraint for the optimization problem and allows the repaired aggregation values to be within the range of $[n+\Delta', n+\Delta]$. In future work, we plan to further study this problem.


Region

Amhara

Complaint: Region: **Amhara**, year: **1983**, mean: **8.00** is too large

Explanation: District: **Ebenat**, year: **1983** should be
count: **0.40,** std: **0.41,** mean: **2.26,**
After repair,: Region: **Amhara**, year: **1983**, mean would be: **7.02**

District	Ebenat
year	1983
mean	8.6666666667
std	2.3094010768
count	3

Explanation 1
Explanation 2
HeatMap: Mean value at District level
HeatMap: Mean value at District level (satellite)
CHIRPS
Scatterplot: Mean and Std of selected District
Bar Chart: Mean and Count of selected District
Detail
Too small
Too large
Drill down
Ebenat
Enbise Sar Midir
Goncha Siso Enese
Kemekem
Shebel Berenta
Mean
Count

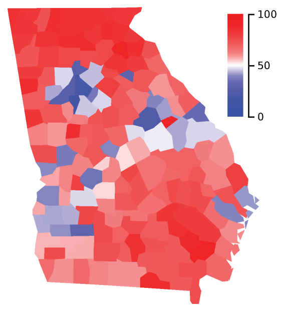
Figure 16: User interface of Reptile

N CASE STUDY: VOTE

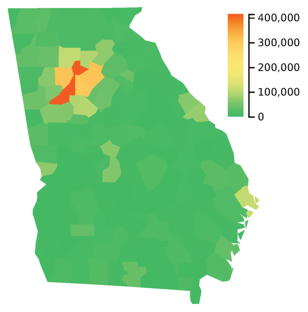
We conduct case study of Vote dataset, which is introduced in Appendix K. We consider a distributive set of two aggregation functions: Percentage of Votes for Donald Trump and total votes. We study Georgia state, which is one of the swing states and Joe Biden wins by a margin about 0.25%. Given the complaint that the Percentage of Votes in the whole state is too low, Reptile is used to find which counties contribute to the loss.

Figure 17a, 17b, 17c and 17d show the Percentage of Votes and total votes of different counties in Georgia in 2016 and 2020. We run Reptile using two models with different features. Model 1 is trained by only default feature, and Model 2 is trained by both default feature and external feature. Figure 17e and 17f show the margin gain of Percentage of Votes after repair by model 1 and model 2. Reptile will recommend counties with larger marginal gain as they better resolve the complaint. For model 1, because it only considers default feature, Reptile mainly detects outliers in the counties of Georgia. Generally, those counties with low Percentage of Votes are deemed with outliers. Model 2 also considers the Percentage of Votes in 2016, which helps explain counties with low Percentage of Votes in 2020. With model 2, Reptile is looking for counties which have abnormal Percentage of Votes or total votes compared to 2016 which after repaired best resolve user complaint. One interpretation of 17f is that it is calculating the change of Percentage of Vote from 2020 to 2016, which is plotted in Figure 17g. While Figure 17f and Figure 17g are correlated, there are obvious differences because Reptile also takes into account the total votes.

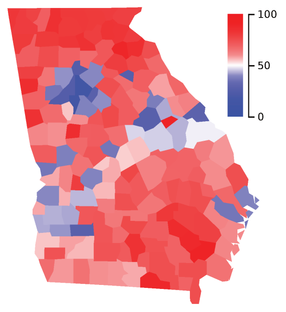
To illustrate the effect of total votes, we manually inject missing records to counties highlighted in Figure 17h by setting total votes to half of its original value. Figure 17i shows the margin gain of data with missing records after repair by model 2. The margin gains of counties with missing records changes depending on its original Percentage of Votes. Reptile combines signals from all aggregation functions and external dataset to make recommendations.



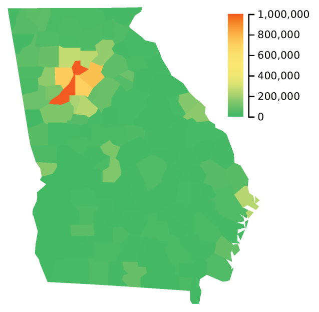
(a) Percentage of Votes in 2016



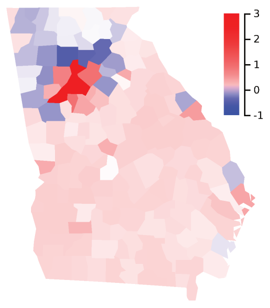
(b) Total Votes in 2016



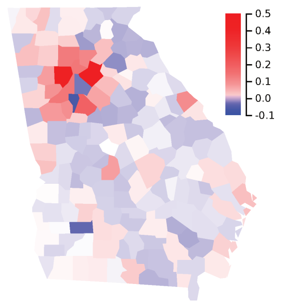
(c) Percentage of Votes in 2020



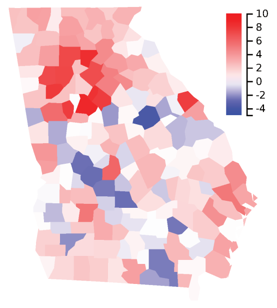
(d) Total Votes in 2020



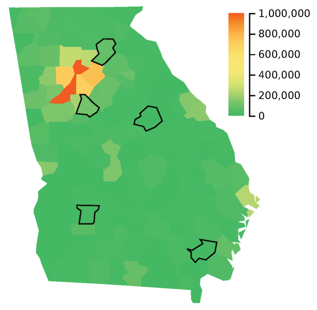
(e) Margin gain after repair by model 1



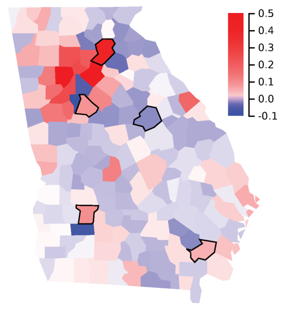
(f) Margin gain after repair by model 2



(g) Percentage of Vote change from 2020 to 2016



(h) Missing records for Total Votes in 2020



(i) Margin gain of data with missing records

Figure 17: Case study of 2020 US presidential election