Reptile: Aggregation-level Explanations for Hierarchical Data

Zezhou Huang
zh2408@columbia.edu
Columbia University

Eugene Wu
ewu@cs.columbia.edu
DSI, Columbia University

ABSTRACT

Recent query explanation systems help users understand anomalies in aggregation results by proposing predicates that describe input records that, if deleted, would resolve the anomalies. However, it can be difficult for users to understand how a predicate was chosen, and these approaches are limited to errors that can be resolved through deletion. In contrast, data errors may be due to group-wise errors, such as missing records or systematic value errors.

This paper presents `Reptile`, an explanation system for hierarchical data. Given an anomalous aggregate query result, `Reptile` recommends the next drill-down attribute, and ranks the drill-down groups based on the extent repairing the group's statistics to its expected values resolves the anomaly. `Reptile` efficiently trains a multi-level model that leverages the data's hierarchy to estimate the expected values, and uses a factorised representation of the feature matrix to remove redundancies due to the data's hierarchical structure. We further extend model training to support factorised data, and develop a suite of optimizations that leverage the data's hierarchical structure. `Reptile` reduces end-to-end runtimes by >6× compared to a Matlab-based implementation, correctly identifies 21/30 data errors in John Hopkin's COVID-19 data, and correctly resolves 20/22 complaints in a user study using data and researchers from Columbia University's Financial Instruments Sector Team.

ACM Reference Format:
Zezhou Huang and Eugene Wu. 2021. Reptile: Aggregation-level Explanations for Hierarchical Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data exploration tools follow the "overview, zoom, then details" analysis pattern [40] 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 (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.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 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	11	1.3
Zata	2.2	9	1.9
...			

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

Sensing Data

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

(c) Auxiliary satellite sensing dataset.

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.

Recent query explanation systems [1, 46, 57] have been proposed to identify predicates over the query input (*explanations*) that, if the records are deleted, will most repair the anomalous result (*complaint*). However, they are limited to deletion-based repairs, and not errors due to missing records, value errors, or anomalous statistics of subsets of the input data. Further, instead of being presented with a (potentially complex) predicate, users often want assistance while incrementally drilling down, so they can verify data at each step. We illustrate with an example based on 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. Rather than read the raw records, the researcher wants to drill-down one level before looking at individual records.*

Although most villages in Ofla reported high severity due to low 1986 rainfall, Darube and Zata have abnormally low means that may contribute to the complaint (Figure 1b). However, Darube's high rainfall in the auxiliary satellite sensing data (Figure 1c) explains the low severity. Zata's low severity is unexplained and thus highlighted when the researcher drills down to villages.

This process is laborious to FIST for many reasons. When there are multiple possible drill-down hierarchies, the choice depends on which of the many resulting groups (e.g., villages) most contributed to the complaint. A group's contribution depends on how much its statistics diverge from its expectation, and the extent that repairing the statistics would address the complaint. However, it is hard to manually estimate a group's expected statistics. Instead, the FIST researchers wish for the ability to submit complaints about anomalous aggregate statistics during data exploration, and be recommended drill-down results to investigate. By repeating this process, they can incrementally zoom into their data and arrive at the data errors.

Existing explanation algorithms fail because each makes assumptions about data errors that do not hold in this context. Density-based approaches such as Smart Drill-Down [24] are designed for count-based complaints, and identify high-cardinality groups; in Example 1, they would return Fala, despite its normal drought severity. Sensitivity-based methods [1, 46, 57] such as Scorpion support complaints over general aggregation functions, but are limited to deletion-based interventions inappropriate for FIST's needs. For instance, deleting Darube village's records would most reduce the standard deviation, but is an incorrect recommendation. Counterbalancing [38] looks for sibling aggregates that offset (counterbalance) the deviation specified in the user's complaint. However, a village's drought is not offset by higher rainfall elsewhere.

The paper presents `Reptile` which tackles the above problem. Given a hierarchical dataset and complaint (e.g., an aggregation query result value is too high or low), `Reptile` recommends the next drill-down attribute and highlights the most relevant groups to examine so the user can understand and verify the recommendations in each step. `Reptile` ranks groups by the extent that repairing their statistics (count, mean, sum, or a user-provided aggregate) to its expected value would resolve the complaint. `Reptile` estimates the expected statistics for each drill-down attribute by fitting a multi-level model[1] to the drill-down's results, and uses the model to exploit the data's hierarchical structure. Users can improve the model by providing additional predictive signals such as custom features (e.g., nearby village severities) or joining with auxiliary datasets (e.g., satellite rainfall estimates).

We address two main 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, which is likely insufficient to train an accurate model. `Reptile` addresses this by using parallel groups (e.g., villages in other districts), and the multi-level model accounts for systematic variation between parent groups (e.g., different districts).

The second challenge is scalable model training. The number of possible models is exponential in the number of attributes, and unrealistic to fully precompute. `Reptile` efficiently trains models online by exploiting functional dependencies inside hierarchies and independence between hierarchies. Instead of materializing a feature matrix exponential in the number of hierarchies, `Reptile` computes a succinct factorised matrix representation [41] that reduces the matrix representation by orders of magnitude. We extend prior work [51, 52], which developed model training procedures over factorised matrices derived from join queries, to matrices based on join-aggregation queries that exhibit fewer redundancies. We further design factorised matrix multiplication operators, and develop a suite of novel work-sharing and caching-based optimizations.

In summary, our contributions are as follows:

- We propose and solve the *complaint-based drill-down problem.*
- We adapt factorized representations to compactly represent the feature matrix, and extend matrix operations to support factorized representations. We develop precomputation, work sharing, and caching optimizations to further accelerate successive drill-down operations.
- On synthetic data, our factorized matrix operations accelerate matrix materialization and gram matrix computations by orders of magnitude, and work-sharing reduces runtimes by 4× over LMFAO [51]. On real-world data, `Reptile` reduces end-to-end performance by 6× as compared to a Lapack-based Matlab implementation.
- We show that `Reptile` robustly identifies multiple classes of errors (missing and duplicate data, systematic value errors) with 70%-100% accuracy, and leverages auxiliary data when it has predictive power; baseline complaint- and outlier-based approaches have 0% to <60% accuracy.
- We evaluated `Reptile` with FIST team members, who explored their drought survey data and submitted 22 complaints. `Reptile` was able able to correctly identify data errors for 20 of them. One failure was inherently ambiguous (team members disagreed about the cause), and `Reptile` partially explained the other.

2 BACKGROUND

2.1 Usage Walkthrough and Architecture

We will use Example 1 to illustrate how the Financial Instruments Sector Team (FIST) uses `Reptile` to identify errors in their farmer-reported drought severity dataset. `Reptile` is initialized with the database as well as metadata about the attribute hierarchies (e.g, geographical and temporal for this example). 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 *Factorizer* 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.

[1] `Reptile` supports custom models, but is optimized for multi-level models.

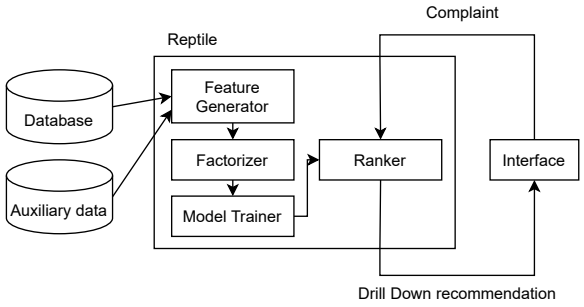


Figure 2: `Reptile` architecture

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 assigns a score to the groups for every possible drill-down (geography and time in this example), and recommends the top-K. The researcher can then submit a village-level complaint to continue drilling down.

2.2 Overview of Factorised Representations

Joins and hierarchical data exhibit redundancy when encoded in a tabular format, and factorised representations [41] (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 not materialize it. Matrix operations in train models reduce to batches of aggregations [51], 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 help remove redundancies due to functional dependencies inside a hierarchy and independence between hierarchies:

Example 2 (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 3 (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 [39].

Consider the aggregation $\gamma_{X_1,...,X_f,\text{COUNT}}(R_1 \bowtie ... \bowtie R_n)$, 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:

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

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

where S_1 and S_2 are the schemas for R and T, $X \in S_1$, $D_1 = \text{Dom}(S_1 \cup S_2)$, and $D_2 = \text{Dom}(S_1 \setminus \{X\})$. Suppose $t = \langle \text{Distinct} = \text{Ofla} \rangle$. 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 4 (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 5 (Early Marginalization). *Let relations* R, T *have schemas* $[A, B]$ *and* $[B, C]$. *Consider the query* $\gamma_{A,\text{COUNT}}(R \bowtie T)$, *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 APPROACH OVERVIEW

3.1 Problem Definition

Given relation $\mathbb{R}$ with attributes $\mathbb{A}$, we assume that all the attributes in $\mathbb{R}$ 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.

Reptile starts with initial view $V = \gamma_{A_{gb},f(A_{agg})}(\mathbb{R})$, where $A_{gb} \subset \mathbb{A}$, $A_{agg} \subset \mathbb{A}$, $A_{gb} \cap A_{agg} = \emptyset$, $f(\cdot)$ is a distributive [20] aggregation function[2] such that, given the partition of $\mathbb{R}$ into J subsets $\mathbb{R}_1, \cdots, \mathbb{R}_J$, there exists function G: $f(\mathbb{R}) = G(f(\mathbb{R}_1), \ldots, f(\mathbb{R}_J))$. Let $t_i \in V$ be an output tuple and $t_i[agg]$ be t_i's aggregation result.

Example 6 (Distributive aggregation function). *Count is a distributive aggregation function. Given the partition of* R *into* J *subsets* $\mathbb{R}_1, \cdots, \mathbb{R}_J$*, we can find* G_{count} *such that* $count(\mathbb{R}) = G_{count}(\{count(\mathbb{R}_1), \cdots, count(\mathbb{R}_J)\}) = \sum_{i=1}^{J}(count(\mathbb{R}_i))$

User can make complaint about tuple $t_c \in V$ in Reptile. Define user complaint as a function $f_{comp} : t \rightarrow \mathbb{R}$ [3] which takes tuple as input and output a value that user aims to minimize. This formulation captures common complaints [5, 38, 46, 57], 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.

Reptile helps user 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 $V = \gamma_{A_{gb},f(A_{agg})}(\mathbb{R})$, drilldown(V, t, H) adds the next strict attribute in hierarchy H to A_{gb} in V and replaces $\mathbb{R}$ by the provenance of t.

Example 7 (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, Reptile tries to repair tuples in drill-down result. Let $f_{repair} : t \rightarrow t$ be a repair function that, given a tuple in the drill-down result, returns a tuple with its expected aggregate statistics. After tuple t' in $V' =$ drilldown(V, t_c, H) is repaired, the complained tuple's aggregation result is also repaired: $t'_c = G(V'/\{t'\} \cup \{f_{repair}(t')\})$

Finally, Reptile proposes one hiearchy $H \in \mathbb{H}$ 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 $f_{comp}(t'_c)$.

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

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

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

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

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

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

Example 8 (Complaint-based Drill-down). *Given the complaint* t_c = (Year : 1986, District : Ofla, count : 62) *in Figure 1, the complaint function is* $f_{comp}(count) = |count - 70|$ *(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 72, then its complaint function would return* 2*, which is preferable.*

Although the user can easily provide t_c and f_{comp}, the repair function f_{repair} is hard to directly express, yet critical to the problem. While users are free to provide a custom repair function, Reptile provides a good default: it fits a multi-level model [17] to estimate the expected aggregate statistics for a given drill-down level. Multi-level models are widely used in fields including sociology [16], demography [48], public health [13], and market sectors [55] to analyze hierarchical data, and improve on linear models by accounting for both the deviation of observations within a group, and the deviation of a group from the other groups. Reptile also provides APIs to easily tune the model (described next).

3.2 Model-based Repair

Reptile identifies erroneous groups by comparing their statistics with its expected statistics based on a model. Models are commonly used to identify and repair numeric errors, and prior works have used log-linear [50], and linear regression models [38]. 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). Below, we illustrate the challenges of a model-based approach using the running example, and then discuss the two techniques Reptile uses to address the challenge.

A key challenge is that there may not be enough groups as the result of a drill-down operation (e.g., villages in Ofla in 1998) to fit an accurate model. We then describe how we use parallel groups to provide more training examples, and use multi-level models to account for variation across the parallel groups.

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 $f(\cdot)$[4] 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.

Using Parallel Groups: Reptile uses the drill-down results for all of the parallel groups (e.g., villages from other districts and years) in the dataset. 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 [16], demography [48], public health [13], and market sectors [55]. Unfortunately, linear models do not take this hierarchical structure into account.

[2] For simplicity, the text assumes a single COUNT aggregation function in Q, however Reptile supports a general distributive set functions, and queries with multiple aggregation functions, as discussed in Appendix A.

[3] 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

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

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:

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

where $\boldsymbol{y}_i$ is the vector of distributive aggregation f(·) 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.

3.3 Tuning the Repair Function

An administrator or end user can programmatically tune the repair function by defining features in $\boldsymbol{X}$. We first describe default, auxiliary, and custom features for $\boldsymbol{X}$, and then describe tuning the random effect matrix $\boldsymbol{Z}$. For simplicity, we assume that each attribute directly translates into a feature. The full details are discussed in Appendix B and Appendix H.

3.3.1 Default Features. Reptile treats all non-aggregation attributes in the drill-down results as categorical. However, naive featurization by hot-one encoding the attributes would exacerbate the dataset sparsity and leads to low prediction accuracy in practice. Instead, we borrow from anomaly detection in multivariate data sets [28] and OLAP data cubes [50], and featurize attributes based on their *main effects* [35]. We replace each categorical attribute value with the median $\boldsymbol{Y}$ of records with the value, and we center and normalize numeric attributes. For instance, if we drill down to (district, village, year) and compute the MEAN statistic for each group, then the feature for the year 1985 would be the median of the mean severities across all villages in 1985. By default, we treat all hierarchy attributes as categorical.

3.3.2 Auxiliary Datasets. Users can reference auxiliary datasets that can be joined with the drill-down results. When the join is possible, Reptile automatically joins and includes the auxiliary measures in the feature matrix. For instance, the village rainfall data in Example 1 is included once Reptile drills down to village. To define an auxiliary dataset, users specify the dataset, join conditions, and its measure attributes.

3.3.3 Custom Features. Users can specify custom per-attribute featurizations. To featurize attribute A, the user defines a function $q(A, \boldsymbol{Y}) \rightarrow \{(a_i, v) | a_i \in A \wedge v \in \mathbb{R}\}$ that takes the attribute values and group statistics as input, and outputs a mapping from attribute value a_i to feature value v. This can express geographical clusters, temporal windows, and other distances. For instance, the previous year's severity may be predictive of this year's.

3.3.4 Random Effect Matrix. The random effects $\boldsymbol{Z}_i$ for the i^{th} cluster are modeled using cluster-specific coefficients b_i. $\boldsymbol{Z}_i$ defines the predictive features to use, and by default uses all features by setting $\boldsymbol{Z}_i = \boldsymbol{X}_i$. Advanced users can tune the random effect matrix by excluding non-predictive features $\boldsymbol{Z}_i$, and Reptile will simply skip those attributes during matrix operations. For example, if users believe that rainfall does not vary by district nor year, they can exclude rainfall and all of its derived features will be ignored.

3.4 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[5] in the form of a tree, where each node is node is either an attribute value, union (∪), or cartesian product (×) (see Section 2.2). 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. [41] for a complete procedure[6]. At a high level, 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 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. Notice that the example has removed the redundant instances of t_1, t_2, and d_1. Appendix C describes the detailed implementation.

Matrix operations loop through the matrix row- or column-wise. These directly correspond to efficient traversals through the f-representation's tree structure.

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

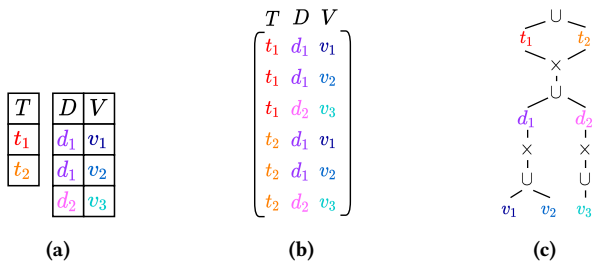


Figure 3: Example dataset with (a) Attribute values organized by hierarchy in attribute order, (b) materialized feature matrix, (c) factorised feature matrix.

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 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 is the primary bottleneck.

Unfortunately, matrix operations do not take factorized matrices as input. This section first decomposes matrix operations into collections of aggregation queries efficiently executable over f-representations. We then develop a suite of work-sharing and caching optimizations to accelerate individual and multi-model training. Our experiments will show that directly operating over f-representations can provide multiple orders of magnitude speedups.

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 [43] in R and statsmodels [53] in Python. In addition, our techniques apply to other algorithms (e.g., Fisher scoring [2], iterative generalized least squares [19]).

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

Prior work [51, 52] focused on factorized matrices derived from join-only queries, and thus only required gram matrix computations for training. In join-only queries, $\boldsymbol{Y}$ can be treated as yet another attribute whose cardinality is independent of the groupby attributes. In contrast, matrices in `Reptile` are derived from join-aggregation queries, so $\boldsymbol{Y}$ is potentially unique for each of an exponential number of groups. This requires our extensions to support left and right multiplications.

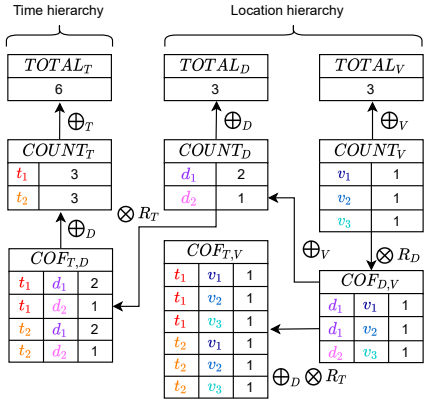


Figure 4: Aggregation results and Multi-query execution

4.2 Factorised Matrix Operations

Prior work [12, 27, 52] decomposes factorized matrix 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 leverage 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, $TOTAL_{\mathrm{A_i}}$, $COUNT_{\mathrm{A_i}}$, $COF_{\mathrm{A_i},\mathrm{A_j}}$, that will be used to define matrix operation outputs. Recall that the feature matrix orders the attributes $\mathrm{A_n}, \ldots, \mathrm{A_1}$ 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.

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

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

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

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

$$\mathrm{i} \in [1, \mathrm{n}], \mathrm{j} \in [1, \mathrm{i}-1]$$

These queries can be naively executed by joining the relations together and then computing the aggregation. The next subsection describes a multi-query optimization that pushes marginalization down, and reuses computation to minimize the intermediate join sizes.

4.2.2 Matrix Operations Using Decomposed Aggregates. We now present the intuition for optimizing the most expensive three matrix operations—gram matrix, left and right multiplication—using data from the running example (Figure 3). The key idea is to use the

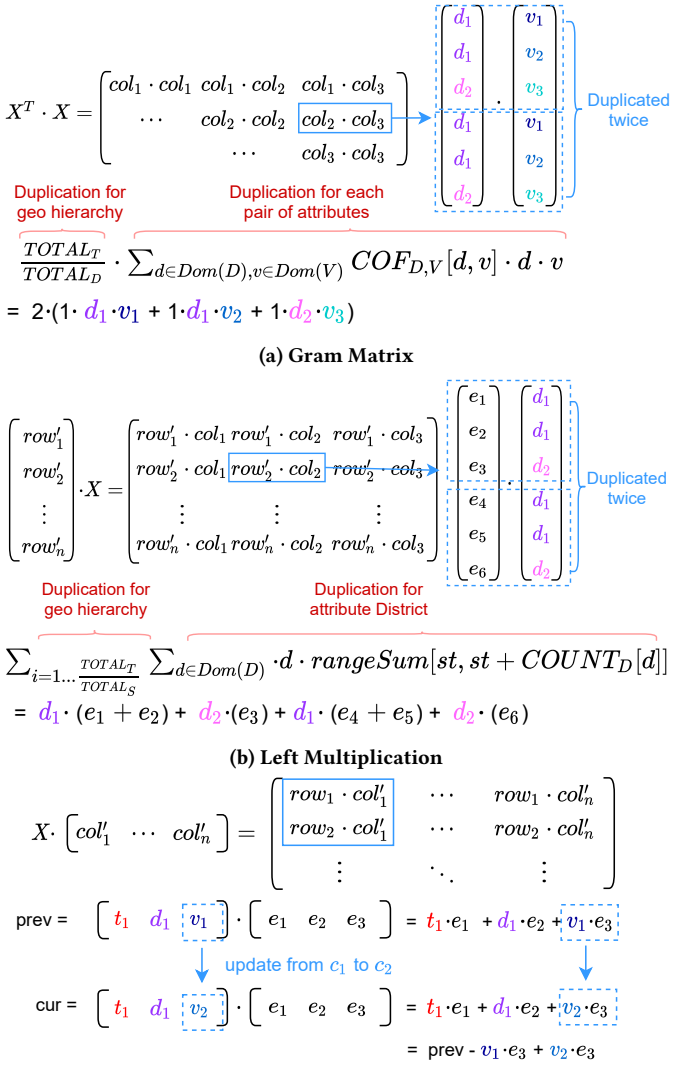


Figure 5: Example matrix operations.

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 [52], but we introduce an optimization based on the data hierarchies.

Gram Matrix: Figure 5a shows the main idea when computing 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 $\text{col}_2 \cdot \text{col}_3$ 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 $\text{row}'_2 \cdot \text{col}_2$, 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., $\text{rangeSum}[0, 2] = e_1 + e_2$). *st* is used to keep track of the start position of row'_2, and is updated for each range summation.

Right Multiplication: This operation uses rows in $\boldsymbol{X}$, so cannot benefit from the techniques above. Figure 5c shows how we leverage the observation that vertically adjacent rows in $\boldsymbol{X}$ have considerable overlap. For instance, the only difference between row_1 and row_2 is the last value ($v_1 \rightarrow v_2$). Thus, output of $\text{row}_2 \cdot \text{col}'_1$ can be incrementally computed from the result of the preceeding row's dot product.

Per-cluster Optimizations: The per-cluster variants use the same algorithms, albeit for the sub-matrices corresponding to the clusters. 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. The details and experiments are in Appendix F.

4.3 Multi-Query Optimization

Early marginalization [52] is applied to push aggregation operator through joins, and work sharing is used to compute 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 requires 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

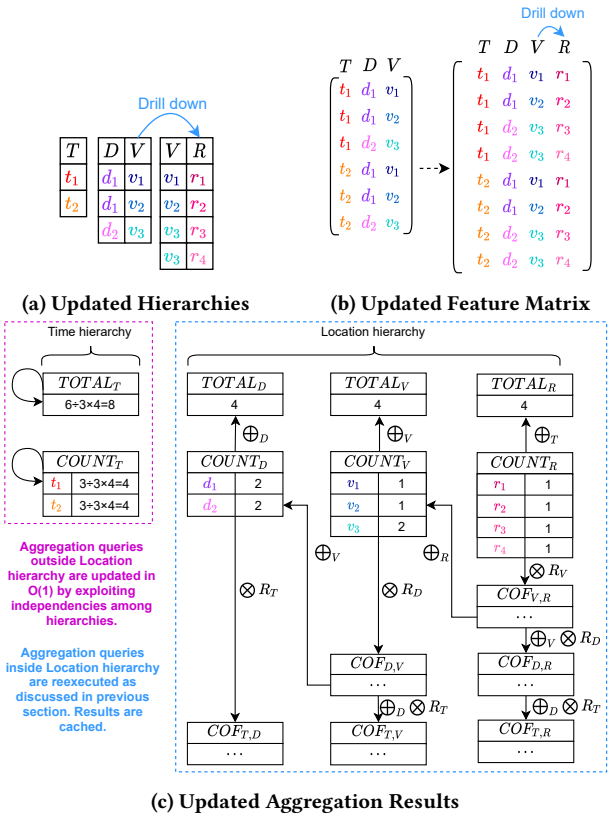


Figure 6: Example Updates after Drill-Down.

$TOTAL_D$ (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.

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. See Appendix J for full details.

Drilldown optimization differs from incremental view maintenance (IVM) for f-representations [39]. 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, Δ ≈ 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++ [7]. 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.

5.1 Performance Evaluation

Given a complaint, `Reptile` enumerates and drills down on each hierarchy, computes decomposed aggregates, builds the (factorized) feature matrix, trains the multi-level model, and ranks the groups. We first evaluate individual steps—the effectiveness of factorized 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 Factorized 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 costs are exponential in d. In contrast, the factorized representation is linear in d.

We measure runtimes for matrix materialization, gram matrix, and left and right multiplication. The former compares the full and factorized matrix construction, while the latter three compares the Lapack[4] implementations over the full feature matrix with `Reptile`'s factorized 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 factorized 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 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

[7] https://github.com/zachary62/Dynamic-F-tree

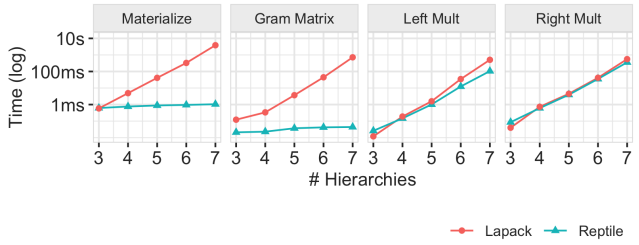


Figure 7: Matrix operation runtimes compared to Lapack-based implementation.

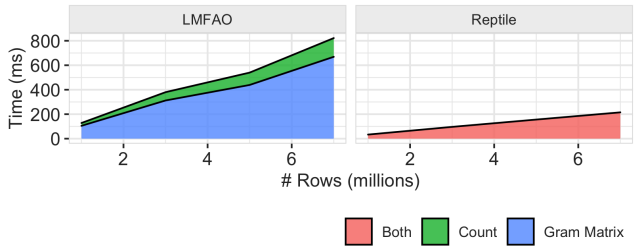


Figure 8: Multi-query execution

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`[51], which is the state-of-art factorised batch aggregation engine implemented in C++[8]. 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 instance, if we drill down from A_2 to A_3, after already drilling down to B_2.

[8]https://github.com/fdbresearch/LMFAO

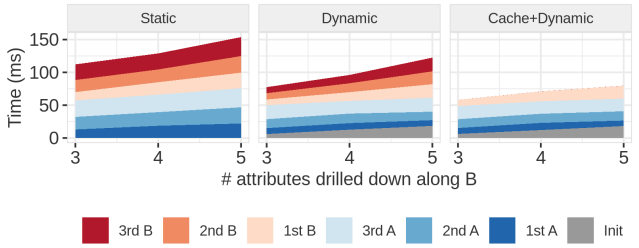


Figure 9: Drill-Down Optimization

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.

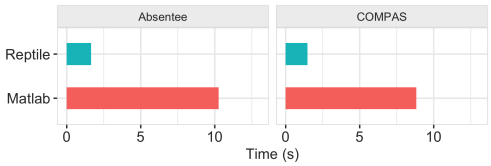


Figure 10: End to end runtime evaluation on real datasets.

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

Absentee[9]: 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[10]: 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

[9]https://www.ncsbe.gov/results-data/absentee-data
[10]https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis

(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. Figure 10 shows Reptile is over 6× faster than Matlab [36], which internally uses Lapack to train over the full materialized feature matrix. This is largely due to avoiding full feature matrix materialization and exploiting its high degree of redundancy through work sharing optimization.

These results also illustrate the limitations of using f-representations in ML. Although matrix materialization and gram matrix are linear in the number of attributes, left and right multiplication remain exponential because the predicted variable $\boldsymbol{y}$ is an aggregate statistic that varies by group. Our evaluation studies the worst-case scenario where the parallel groups include all exponential number of drill-down groups (even empty groups). A simple optimization may be to only sample or truncate the number of parallel groups to train over, or to prune empty groups.

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 work. 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. We generate a dataset with one dimension attribute (i.e., one hierarchy) that has 100 unique values (and thus 100 groups), and one measure for computing aggregates. The number of rows in each group is drawn from a normal distribution $\mathcal{N}(100, 20)$, and each measure value is drawn from $\mathcal{N}(100, 20)$. In each experiment, we generate 1000 datasets and report the average accuracy of the top group.

Auxiliary Data: Reptile is able to combine auxiliary data provided by domain experts which has a (potentially weak) correlation to the correct aggregate statistics. We simulate this by generating one auxiliary table for each aggregate statistic (COUNT, MEAN, STD), where STD is only used when evaluating the Raw condition described below. The auxiliary table contains the same dimension attribute in original dataset, 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 Iman and Conover [23].

Error Generation: We randomly chose a group to be erroneous, and introduced different classes of errors: missing/duplicate records to change the COUNT statistics, and data drift [6] to change the MEAN statistic. For the former, 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.

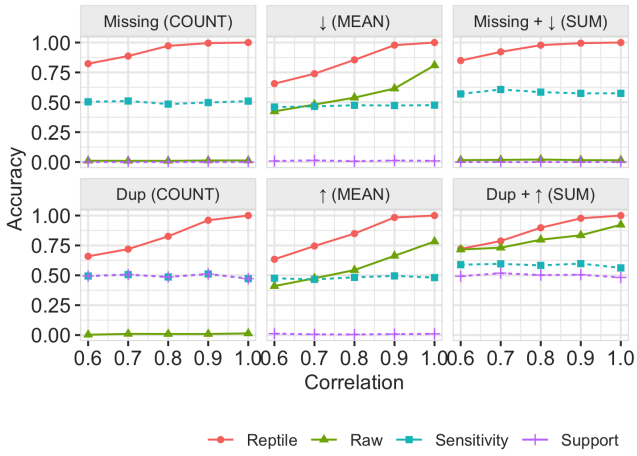


Figure 11: Accuracy comparison with naive approaches and prior work. ↑ is Increase, ↓ is Decrease, and Dup is Duplication. The complained aggregation is in the parentheses.

Approaches: Five approaches are used to identify the erroneous group: Reptile, Outlier, Raw, Sensitivity [57] and Support. Reptile utilizes the auxiliary data to repair the dataset and recommends one group which, after repaired, best resolves the complaint. Outlier ignores the complaint and simply returns the group whose statistics most deviates from the model's prediction. Raw is a record-level bottom up approach based on Winsorization [29]. For each drill-down group, it computes the mean and standard deviation of the measure attribute within the group, and clips each input row's measure to [MEAN - STD, MEAN + STD]. Raw then returns the group whose clipping-based repairs best resolves the complaint. Finally, we compare with two prior explanation approaches: Sensitivity is based on interventional deletions [57], and recommends the group which, after deleting all rows, best resolves the complaint. Support is a density-based approach that returns the fraction of rows in a drill-down row. It is commonly used as a pruning criterion in explanation systems [1]. In this experiment, this amounts to recommending the group with the largest COUNT (i.e., support) as there is only one dimension attribute.

5.2.2 Baselines. We first evaluate Reptile against Raw, and the two prior approaches (Outlier is defered next). Figure 11 varies the correlation of the auxiliary data (x-axis) for the different error types (columns). Raw fails to detect missing/duplicated records because repairing at raw data level can't capture these errors. Raw performs well for Duplication + Increase but poorly for Missing + Decrease because SUM is sensitive to group with a larger number of rows: Raw searches for group which after drifting its values back best resolves the complaint, and for group with duplicated rows, the drift has more impact to SUM so that it is more likely to be recommended. Sensitivity and Support are flat because they do not leverage auxiliary data. Sensitivity fails to leverage auxiliary data, so its recommendation is less reliable. Support only performs well

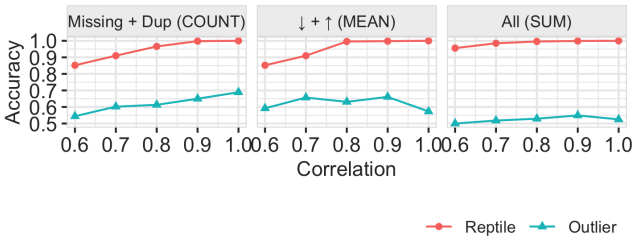
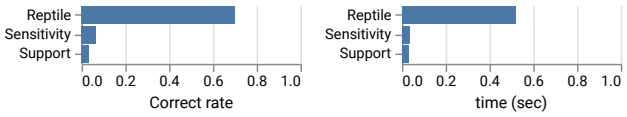


Figure 12: Accuracy comparison for multiple errors.



(a) Avg. accuracy of top result. **(b) Average Runtime**

Figure 13: COVID-19 Case Study

under duplication because it is density-based and is designed for the complaint "COUNT is high". Reptile is considerably and consistently more accurate, and successfully leverages the auxiliary data even when the correlation is weak.

5.2.3 Complaint Ablation. We now compare Outlier with Reptile to study the value of leveraging complaints to distinguish between different possible errors. To do so, we choose two groups whose error affects the complaint (true errors), and one group whose error does not (false positive). We generate three conditions: **Missing + Duplication** corrupts two groups with missing records (the correct errors), and one group with duplication. The complaint is "COUNT is low". ↓ + ↑ introduces data drift to two groups by decreasing their measure values (the correct errors) and one group by increasing the measure values. The complaint is "MEAN is low". **All** corrupts two groups by decreasing the measure values and causing missing records. For the false-positive group, we increase the measure values and introduce duplicates. The complaint is "SUM is low".

Figure 12 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 Outlier hovers between 50-70% accuracy because, while Outlier is able to identify three imputed groups, it cannot distinguish between them. Given that only two of them are correct, the accuracy of Outlier is bounded by 66%.

5.3 Case Study: COVID-19

The COVID-19 data [14] 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 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: The dataset's Github issues report a variety of data errors that have been confirmed and resolved, and we use errors resolved between 12/2/2020 and 1/27/2021 as ground truths for evaluation. We generate a corrupt dataset for each issue, submit a complaint, and compare methods to identify the erroneous location.

Most issues are due to missing data on a specific day, which causes underreporting. Others may be due to backlogged reports (e.g., on days n−m) that are totaled and reported on day m+1, or due to changes in a location's reporting methodology[11]. We use 16 (14) issues from the US (global) datasets and construct a complaint for each issue. To do so, we first filter by the complaint's day, aggregate the total statistics at the immediately higher geographical level (e.g, if the error is in New York, then we compute the US aggregate), and then specify whether the result is too high or too low based on the ground truth. 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 Sensitivity [57] and Support (described in Section 5.2).

Results and Error Analysis: Figure 13 shows that although Reptile takes longer in order to fit models (≈0.5s), it is considerably more accurate (70%) as compared to the baselines (6.6% for Sensitivity, 3.3% for Support). We conducted an error analysis of the 9 errors that Reptile did not identify. The first type (5 issues) is 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 included all Quebec data in the experiment. The second type (4 issues) are subtle issues smaller than the natural variation in the data, and unlikely to result in complaints. For example, on 12/18/2020, Washinton state submitted 21,308 instead of 21,038. Appendix L discusses the experiment in more detail.

5.4 Case Study: FIST

The Columbia University Financial Instruments Sector Team (FIST) group collects Ethiopian farmer-reported drought data to design drought insurance [42]. 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[12] 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

[11] https://www.azdhs.gov/preparedness/epidemiology-disease-control/infectious-disease-epidemiology/covid-19/dashboards/

[12] We attempted to recruit novice users, however they could not interpret the domain-specific data.

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 [9] to find violations, while quantitative error detection often relies on statistical methods (e.g., outlier detection [5, 22, 30, 45] or explicit error-prediction models [21, 31, 34]). Reptile combines a complaint-based approach [1, 8, 38, 46, 57] 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 [9, 58], and can leverage signals (e.g., past repairs [56], knowledge bases [10]). The solution space is large, so human involvement is crucial. Spreadsheet interfaces [25, 44] and visualizations [25, 32, 57] help users identify potential errors and help guide repairs. Interfaces such as Profiler [26] run a library of common error detectors and embed the errors directly in the visualization interface.

Model-based repairs estimate the correct value of an error. ERACER [37] uses graphical models that combine convolution and regression models to repair raw data tuples. Active learning approaches [33, 54, 59] ask users to verify whether repair candidates are correct. Daisy [18] 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. Finally, techniques such as unknown unknows [11] 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 [1, 8, 46, 57], density-based [24, 47, 49], counterbalance [38]), and typically focus on deletion-based interventions. Reptile ranks drill-down groups based on how much repairing their aggregate statistics, as learned by a multi-level model, would resolve the complaint. Repairing at aggregation level enables Reptile to combine predictive signals and uncover a broader range of errors like missing records which previous metrics fail to detect.

The hierarchical density attribution problem [15, 49] returns a set of non-overlapping subgroups that account for the largest mass of the total density. Ruhl et al. [47] extends this from a single hierarchy to a product of trees (i.e., overlapping hierarchies) and shows that this problem is NP-hard. Joglekar et al. [24] is restricted to count-based densities and leverages its submodular structure to design a greedy solution to recommending sets of drill-down groups. Reptile is designed for a single hierarchy and supports more complex aggregation functions, and returns a ranked list of drill-down groups rather than an optimal set.

Factorised Representation: Factorised Representation [41] reduces redundancies due to functional dependencies, and has been used to optimize model training (linear regression [52], decision tree [27] and Rk-mean [12]) over factorised matrices derived from join queries. Reptile extends prior work [51, 52] 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

We presented Reptile, which helps users iteratively identify and repair errors in the output of aggregation queries. Reptile supports the "Overview, zoom, details-on-demand" analysis pattern common in visual analysis by recommending drill-down operations and highlighting groups in the drill-down results that most contributed to the user's reported data error. Reptile trains a model to estimate each group's expected aggregate statistics, and measures the extent that the complaint is resolved by repairing the group statistic to its expectation. 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 Ethipoian farmers.

REFERENCES

[1] F. Abuzaid, P. Kraft, S. Suri, E. Gan, E. Xu, A. Shenoy, A. Ananthanarayan, J. Sheu, E. Meijer, X. Wu, et al. Diff: a relational interface for large-scale data explanation. *The VLDB Journal*, pages 1–26, 2020.

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

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

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

[5] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556, 2017.

[6] J. P. Barddal, H. M. Gomes, F. Enembreck, and B. Pfahringer. A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. *Journal of Systems and Software*, 127:278–294, 2017.

[7] K. P. Burnham and D. R. Anderson. Multimodel inference: understanding aic and bic in model selection. *Sociological methods & research*, 33(2):261–304, 2004.

[8] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 445–456, 2014.

[9] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[10] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. Katara: reliable data cleaning with knowledge bases and crowdsourcing. *Proceedings of the VLDB Endowment*, 8(12):1952–1955, 2015.

[11] Y. Chung, M. L. Mortensen, C. Binnig, and T. Kraska. Estimating the impact of unknown unknowns on aggregate query results. *ACM Transactions on Database Systems (TODS)*, 43(1):1–37, 2018.

[12] R. Curtin, B. Moseley, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*, pages 2742–2752, 2020.

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

[14] E. Dong, H. Du, and L. Gardner. An interactive web-based dashboard to track covid-19 in real time. *The Lancet infectious diseases*, 20(5):533–534, 2020.

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

[16] R. M. Fernandez and J. C. Kulik. A multilevel model of life satisfaction: Effects of individual characteristics and neighborhood composition. *American Sociological Review*, pages 840–850, 1981.

[17] A. Gelman and J. Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[18] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 805–815, 2020.

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

[20] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[21] A. Heidari, J. McGrath, I. F. Ilyas, and T. Rekatsinas. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*, pages 829–846, 2019.

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

[23] R. L. Iman and W.-J. Conover. A distribution-free approach to inducing rank correlation among input variables. *Communications in Statistics-Simulation and Computation*, 11(3):311–334, 1982.

[24] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. Smart drill-down: A new data exploration operator. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 8, page 1928. NIH Public Access, 2015.

[25] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372, 2011.

[26] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 547–554, 2012.

[27] L. Kobis. Learning decision trees over factorized joins. 2017.

[28] J. Laurikkala, M. Juhola, E. Kentala, N. Lavrac, S. Miksch, and B. Kavsek. Informal identification of outliers in medical data. In *Fifth international workshop on intelligent data analysis in medicine and pharmacology*, volume 1, pages 20–24. Citeseer, 2000.

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

[30] F. Liu, K. Ting, and Z. Zhou. Isolation forest. *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[31] Z. Liu, Z. Zhou, and T. Rekatsinas. Picket: Self-supervised data diagnostics for ml pipelines. *arXiv preprint arXiv:2006.04730*, 2020.

[32] Y. Luo, C. Chai, X. Qin, N. Tang, and G. Li. Visclean: Interactive cleaning for progressive visualization. *Proc. VLDB Endow.*, 13:2821–2824, 2020.

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

[34] M. Mahdavi, Z. Abedjan, R. Castro Fernandez, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 865–882, 2019.

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

[36] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[37] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 75–86, 2010.

[38] Z. Miao, Q. Zeng, B. Glavic, and S. Roy. Going beyond provenance: Explaining query answers with pattern-based counterbalances. In *Proceedings of the 2019 International Conference on Management of Data*, pages 485–502, 2019.

[39] M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*, pages 365–380, 2018.

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

[41] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

[42] D. Osgood, B. Powell, R. Diro, C. Farah, M. Enenkel, M. E. Brown, G. Husak, S. L. Blakeley, L. Hoffman, and J. L. McCarty. Farmer perception, recollection, and remote sensing in weather index insurance: An ethiopia case study. *Remote Sensing*, 10(12):1887, 2018.

[43] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[44] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

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

[46] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1579–1590, 2014.

[47] M. Ruhl, M. Sundararajan, and Q. Yan. The cascading analysts algorithm. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1083–1096, 2018.

[48] J. M. Sacco and N. Schmitt. A dynamic multilevel model of demographic diversity and misfit effects. *Journal of Applied Psychology*, 90(2):203, 2005.

[49] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, volume 99, pages 7–10. Citeseer, 1999.

[50] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *International Conference on Extending Database Technology*, pages 168–182. Springer, 1998.

[51] M. Schleich and D. Olteanu. Lmfao: An engine for batches of group-by aggregates. *arXiv preprint arXiv:2008.08657*, 2020.

[52] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18, 2016.

[53] S. Seabold and J. Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[54] S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J.-A. Quiane-Ruiz, and N. Tang. Uguide: User-guided discovery of fd-detectable errors. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1385–1397, 2017.

[55] S. Van de Walle, B. Steijn, and S. Jilke. 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):833–855, 2015.

[56] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *2014 IEEE 30th international conference on data engineering*, pages 244–255. IEEE, 2014.

[57] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.

[58] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. 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*, pages 553–564, 2013.

[59] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *arXiv preprint arXiv:1103.3103*, 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 [20] 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:

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

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

$$G_{\text{std}}(\mathbb{F}_{\text{agg}}(\text{R}_1), \cdots, \mathbb{F}_{\text{agg}}(\text{R}_\mathcal{J})) = \sqrt{\frac{\sum_{\text{j}=1}^{\mathcal{J}}(\text{F}_{\text{count}}(\text{R}_j) - 1) \cdot \text{F}_{\text{std}}^2(\text{R}_j) + \sum_{\text{j}=1}^{\mathcal{J}} \text{F}_{\text{count}}(\text{R}_j) \cdot (G_{\text{mean}} - \text{F}_{\text{mean}}(\text{R}_j))^2}{G_{\text{count}} - 1}}$$

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 3.4). In Figure 14, given hierarchies in Figure 14a with order: Time and Location, attribute matrix is shown in Figure 14c.

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 14, all applicable features are shown in Figure 14b and feature matrix is shown in Figure 14d.

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 14, 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 $\sum_{\text{a}\in\text{Dom(T)}} COUNT_\text{T}(\text{a}) \cdot \text{f}^\text{a}(\text{a}) = 3\text{f}^\text{a}_{\text{t}_1} + 3\text{f}^\text{a}_{\text{t}_2}$. The isolation of attribute from feature can simplify the problem and improve performance because attribute matrix is smaller than feature matrix.

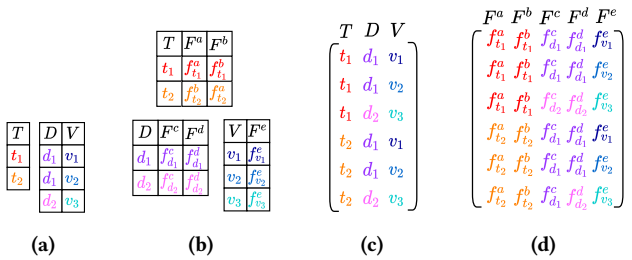


Figure 14: Example dataset with (a) hierarchies, (b) features, (c) attribute matrix, and (d) feature matrix.

C FACTORIZER

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

C.1 Storage

We discuss how *Factorizer* stores factorised feature matrix. *Factorizer* 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, *Factorizer* normalizes relations to BCNF, sort them according to attribute order, and stores relations using sorted map. For example data in Figure 14 with attribute order "Time (T), District (D), City (V)", *Factorizer* 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). *Factorizer* 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

Factorizer presents interface of relation and row iterator.

Relation: Given attribute, *Factorizer* 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, factorizer returns $\text{R}_\text{i}[\text{A}_\text{i}]$ which enumerates all attribute values in A_i. Otherwise, factorizer 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 14, *Factorizer* 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, *Factorizer* exploits the fact that the rows in attribute matrix are sorted by the attribute order and the difference between rows is relatively small. *Factorizer* 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*, $itr_{A_{i+1}}$ 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
itr_{A_i} := current iterator for attribute A_i;
nextValue := itr_{A_i}.next();
update[A_i] := nextValue;
if current attribute value ∈ end and A_i ≠ root then
    next(parent(A_i), update);
if !itr_{A_i}.hasNext() then
    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 $\hat{\mathbf{b}}_i \cdot \hat{\mathbf{b}}_i^{\mathrm{T}}$:

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

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

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

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

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:

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

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

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

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 m × n:

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

This could be avoided by reordering matrix multiplications:

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

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 14c, 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 *Factorizer* 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 $\in$ *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 ($\boldsymbol{X}_i^T \cdot \boldsymbol{X}_i$, $\boldsymbol{X}_i \cdot \boldsymbol{C}_i$, $\boldsymbol{D}_i \cdot \boldsymbol{X}_i$ 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 $\boldsymbol{X}_i^T \cdot \boldsymbol{X}_i$ 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 = Σ_{j=0}^{number of inter cluster attributes} r_inter[j] × 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

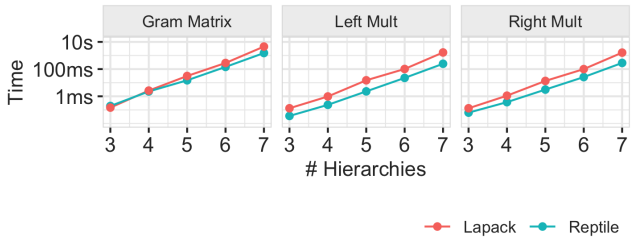


Figure 15: Matrix operation over clusters runtimes compared to Lapack-based implementation.

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 15 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 $\frac{TOTAL_{A_d}}{TOTAL_{A_{first}}} \cdot \sum_{(a_1,...,a_k)\in Dom(\mathbb{A})} COF\,[(a_1, ..., a_k)] \cdot f_i(\sigma_{\mathbb{A}_p}(a_1, ..., a_k)) \cdot f_j(\sigma_{\mathbb{A}_q}(a_1, ..., a_k))$;

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 $(a_1, ..., a_k) \in Dom(\mathbb{A}_p)$ *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:

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

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 $k > j$:

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

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

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

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

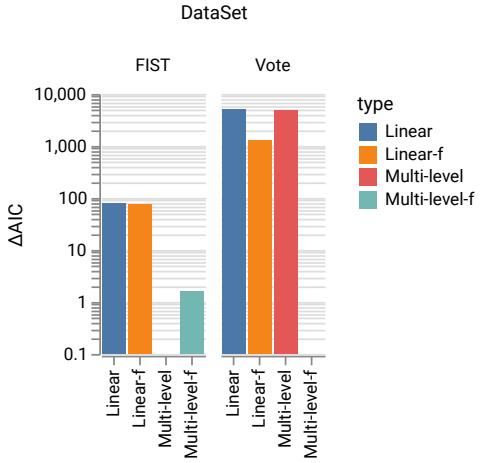


Figure 16: Model evaluation

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) [3], 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 [7].

The result of model evaluation is shown in Figure 16. 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

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
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`. Prevalent errors are highlighted with ★.

ID	Issue	RP	ST	SP
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.

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 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 17 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. [24] 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.

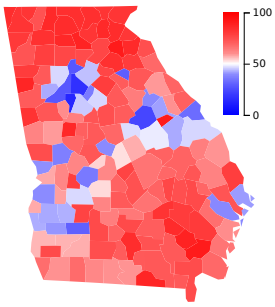
Figure 17: User interface of Reptile

N CASE STUDY: VOTE

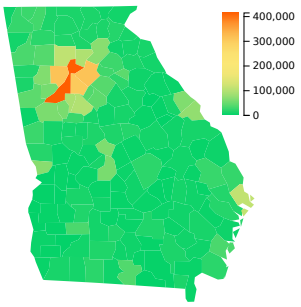
We conduct case study of Vote dataset, which is introduced in Appendix K. We consider a distributive set of two aggregation functions: Percentage of Votes for Donald Trump and total votes. We study Georgia state, which is one of the swing states and Joe Biden wins by a margin about 0.25%. Given the complaint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

Figure 18a, 18b, 18c and 18d 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 18e and 18f 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 18f is that it is calculating the change of Percentage of Vote from 2020 to 2016, which is plotted in Figure 18g. While Figure 18f and Figure 18g are correlated, there are obvious differences because Reptile also takes into account the total votes.

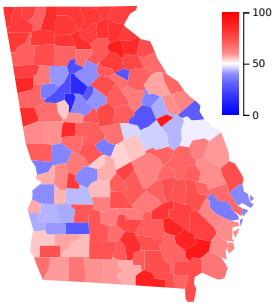
To illustrate the effect of total votes, we manually inject missing records to counties highlighted in Figure 18h by setting total votes to half of its original value. Figure 18i shows the margin gain of data with missing records after repair by model 2. The margin gains of counties with missing records changes depending on its original Percentage of Votes. Reptile combines signals from all aggregation functions and external dataset to make recommendations.



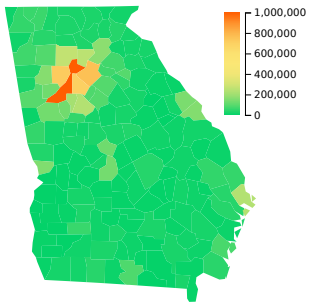
(a) Percentage of Votes in 2016



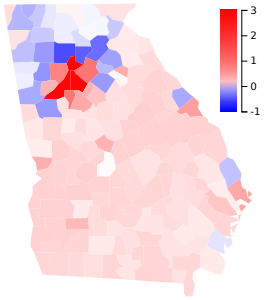
(b) Total Votes in 2016



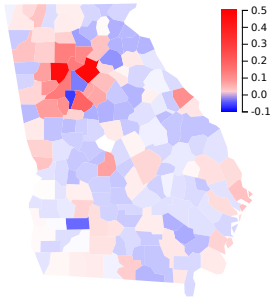
(c) Percentage of Votes in 2020



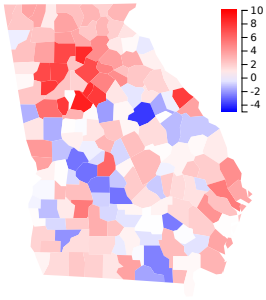
(d) Total Votes in 2020



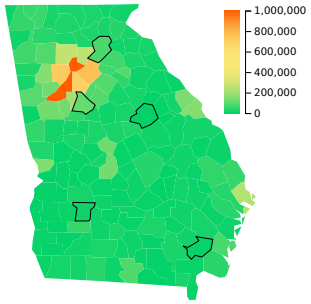
(e) Margin gain after repair by model 1



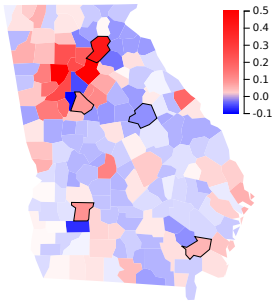
(f) Margin gain after repair by model 2



(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 18: Case study of 2020 US presidential election