Diego Escondrillas → A20528684

Syed Wali Uddin Quadri → A20554645

REPTILE: Reptile: Aggregation-level Explanations for Hierarchical Data

Abstract:

The paper introduces Reptile, an explanation system designed for hierarchical data, addressing limitations in existing query explanation systems. While traditional systems propose predicates for anomaly resolution through record deletion, they struggle with user interpretability and are confined to deletion-based error resolution. Reptile takes a different approach, offering explanations for group-wise errors, such as missing records or systematic value errors in hierarchical data. It recommends the next drill-down attribute and ranks drill-down groups based on the effectiveness of repairing group statistics in resolving anomalies/complaints. A complaint could be described as an anomalous result, an inconsistency or a violation of certain properties within the hierarchy.

Reptile employs an efficient multi-level model, leveraging data hierarchy for expected value estimation, and utilizes a factorized representation of the feature matrix to eliminate redundancies from hierarchical data. With optimizations, it significantly reduces runtimes, outperforming a Matlab-based implementation. Evaluation on COVID-19 data and a user study with Columbia University's Financial Instruments Sector Team demonstrates Reptile's effectiveness in identifying and resolving data errors

Introduction:

When dealing with hierarchical datasets, there could be various ways to drill down into the data to investigate issues or complaints. The choice of which path to take depends on identifying groups (e.g., villages in a dataset) that significantly contribute to a particular complaint. The level of contribution is determined by how much the statistics of a group deviate from what is expected, and fixing these statistics should ideally address the complaint.

However, manually estimating what statistics to expect from a group can be challenging. This is where Reptile takes place. It recommends specific drill-down results to investigate further. Drilling down in hierarchical datasets refers to the process of navigating deeper into the hierarchy to view more detailed or granular information.

When faced with a hierarchical dataset and a specific complaint (for instance, when an aggregated query result is exceptionally high or low), Reptile aims to guide the user in exploring the data. It does this by recommending the next attribute to drill down into and highlighting the most relevant groups for examination. The goal is to assist the user in comprehending and validating these recommendations at each step of the exploration process.

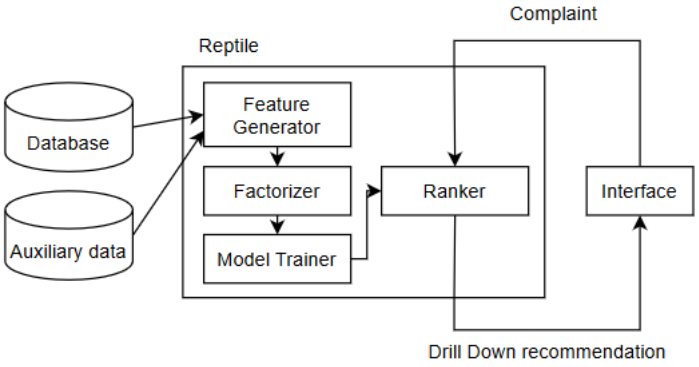
Reptile determines the relevance of groups by ranking them based on how much correcting their statistics (count, mean, sum, or a user-defined aggregate) to their expected values would potentially resolve the specific complaint. Reptile estimates the expected statistics for each

drill-down attribute by fitting a multi-level model to the drill-down's results, and uses the model to exploit the data's hierarchical structure.

A multi-level model is a statistical model that considers different levels or layers of variation in the data. Multi-level models are used 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.

Background:

Reptile follows the following architecture:



As it can be seen in the left side of the image, Reptile takes 2 inputs: Database and Auxiliary Data. This is because when a user submits a complaint, he can also provide auxiliary data that could help Reptile solve the complaint.

The Factorizer stores the features in an efficient factorised representation, and the Model Trainer fits a predictive model to estimate the statistics for each group in the next candidate drill-down.

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, and recommends the top-K.

The reason for which factorized representations are used is that when data is stored hierarchically, some of the data is redundant when encoded in a tabular format. However, factorized representations remove this redundancy. F-representations help remove redundancies due to functional dependencies inside a hierarchy

Approach Overview:

A complaint example could be that the aggregation of a certain tuple is too high or too low. On the other hand, the complaint function fcomp(t) is used to specify a value that the tuple should have.

After drilling down, Reptile tries to repair tuples in the drill-down result. Reptile does this by using a repair function that, given a tuple in the drill-down result, returns a tuple with its expected aggregate statistics. After the tuple is repaired after the drill-down operation, the complained tuple's aggregation result is also repaired (t'c).

Finally, Reptile proposes one hiearchy H $\in$ H to drill-down, and one tuple t $\in$ drilldown (V, tc, H) such that "fixing" the t's group statistics would minimize user complaint fcomp(t'c).

The users can submit to Reptile a repair function apart from the complaint and the complaint function if they want. However, Reptile provides a good default repair function which fits a multi-level model to estimate the expected aggregate statistics for a given drill-down level.

The main challenge of using a model-based approach is that there may not be enough groups as the result of a drill-down operation to fit an accurate model. However, Reptile uses parallel groups to provide more training examples and uses multi-level models to account for variation across the parallel groups.

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.

Tuning the Repair Function:
As mentioned above, users that submit the complaint can also submit 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. To define an auxiliary dataset, users specify the dataset, join conditions, and its measure attributes.

Factorized feature matrix:

As it was mentioned, rather than materialize the full matrix, Reptile constructs a factorised matrix representation. This factorized matrix is represented in the form of a tree, where each node is either an attribute value, union (U), or cartesian product (×). In order to get the tree shape, an attribute ordering must be established.

The attributes are ordered by selecting an ordering of the hierarchies, and within each hierarchy, the attributes are ordered from least to most specific (in the example Village is more specific than District). The main restriction is that the hierarchy that we are drilling down should be ordered last.

Lets assume the following example:

The figure below 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 X (Figure 3b) is computed as the cross product between the two hierarchy tables. Note the redundancy across the hierarchies (ti is replicated), and within the Geo hierarchy (d1 is replicated).

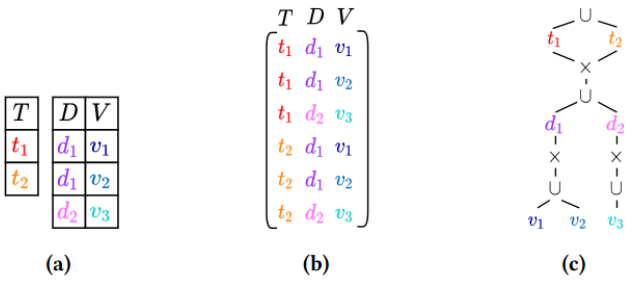


Figure 3: Example dataset with (a) Attribute values organized by hierarchy in attribute order, (b) materialized feature matrix, (c) factorised feature matrix.

The way by which we get from (b) to (c) is the following:

At a high level, each attribute corresponds to one level of the tree, and Ai's level is directly above Aj's if Ai directly precedes Aj in the attribute order.

Each node (e.g., t1) 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 (c), "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.

Details and optimizations:

Reptile, in each cycle of its operation, suggests the next level of detail to explore in the dataset and provides a list of the most relevant groups related to this exploration. This involves constructing a structured representation of features (factorized feature matrix), creating a multi-level model, predicting expected values for each group using this model, and then prioritizing groups based on how fixing their statistical values would address the user's complaint. The training of the model is highlighted as a crucial step in this process, as it's the primary factor determining the efficiency of Reptile's recommendations.

Reptile deals with factorized matrices, which are not directly compatible with standard matrix operations. This section breaks down matrix operations into sets of aggregation queries that can be efficiently executed on factorized representations (f-representations). This approach allows for substantial speedups, as it avoids the need to materialize the entire matrix.

EM-based model training:

The multi-level model used by Reptile is trained using the Expectation-Maximization (EM) algorithm, a standard technique for such models. This involves various types of matrix multiplications: gram matrix ($X^T \cdot X$), right multiplication ($X \cdot A$), and left multiplication ($B \cdot X$) (X is the full matrix). Reptile outlines how these operations are performed on factorized matrices. The goal is to avoid the computational cost of creating the full X matrix and instead operate directly on the f-representation.

Factorized matrix operations:

Previous work decomposes factorized matrix operations into a series of aggregation queries. Reptile extends this idea to support not only gram matrix computations but also left and right multiplications. These operations are essential for training models based on the unique characteristics of Reptile's datasets.

Note: Attributes to the right of Ai means they appear hierarchically to the right of the attribute Ai (more specific). So, when referring to computations or aggregations involving "all attributes to the right of Ai," it means considering the attributes that are more specific in the hierarchy than Ai, including Ai itself.

The most expensive matrix operations are gram matrix, left and right multiplication.

- Gram matrix ($X^T \cdot X$) Optimization:

    - Reptile optimizes the computation of the gram matrix by quantifying redundancy using decomposed aggregates.
- Left Multiplication (B · X) Optimization:
    - Left multiplication involves iterating over district values and multiplying them by corresponding elements in a row.
    - Reptile preprocesses the row by computing the prefix sum for fast range summations.
- Right Multiplication (X · A) Optimization:
    - Right multiplication leverages the observation that vertically adjacent rows in X have considerable overlap.
    - Incremental computation is used, allowing the result of the preceding row's dot product to contribute to the next row.
- Per-Cluster Optimizations:
    - The per-cluster variants of operations use similar algorithms for sub-matrices corresponding to clusters.
    - Clusters correspond to siblings in the f-representation, and they benefit from work-sharing optimizations.
    - Caching contributes to the matrix operation's output for shared attributes among clusters.

As it can be seen, these optimizations contribute to the efficiency and speed of multi-model training in the context of hierarchical datasets.

Drill-down optimization:

When a user drills down a hierarchy, it means they are moving from a more general level to a more specific one.

Regarding the drilling down process, each drill-down operation adds additional columns to the factorized feature matrix, corresponding to the next attribute in the hierarchy.

After a drill-down, the multiplicities (counts) of the preceding attributes might change. Reptile suggests an efficient update mechanism for these multiplicities based on the observation that attributes between different hierarchies are independent.

The results of decomposed aggregates for attributes in the other hierarchies are cached to accelerate subsequent calls.

Even though the user may pick one drill-down hierarchy, Reptile needs to re-evaluate all hierarchies in the next call, and caching helps speed up this process.

Putting It All Together:

In each iteration, Reptile follows a series of operations to recommend drill-down results that address the user's complaint. Here's a breakdown of the steps:

1. Construct Factorized Feature Matrix:

    a. For each candidate hierarchy H, Reptile constructs the factorized feature matrix after the drill-down operation. This matrix represents the dataset in a factorized form, which is efficient for subsequent computations.
2. Recompute Decomposed Aggregates:
    a. The algorithm recomputes the decomposed aggregates for the attributes in H using multi-query optimizations. This involves efficiently calculating counts and other statistics associated with the attributes in the hierarchy.
3. Update Remaining Decomposed Aggregates:
    a. Reptile updates each of the remaining decomposed aggregates in constant time. This is possible due to the independence of attributes between different hierarchies. The algorithm efficiently adjusts counts and multiplicities based on the changes introduced by the drill-down.
4. Translate EM into Matrix Operations:
    a. The algorithm translates the Expectation-Maximization (EM) algorithm into matrix operations. EM is used for fitting the multi-level model's parameters via maximum likelihood estimation. Matrix operations are executed until parameter convergence.
5. Repair Drill-Down Groups:
    a. Using the trained model, Reptile repairs each drill-down group based on the model's prediction. Repairing involves adjusting statistics, such as counts or means, to bring them closer to their expected values.
6. Incrementally Update Complaint:
    a. Reptile incrementally updates the user's complaint to check the extent to which it is resolved. This involves evaluating the impact of the repairs on the specific complaint and determining whether further iterations are needed.

Experiments:

The experiments conducted in the paper aim to evaluate the effectiveness of the proposed optimizations in Reptile, focusing on runtime performance and the ability to identify group-wise data errors. The evaluation involves synthetic data, complaints, and real-world case studies.

The case studies used to conduct the experiments are based on known errors in COVID-19 data and 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.

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.

Synthetic datasets vary in the number of hierarchies (d = 3), attributes in each hierarchy (t = 3), and unique values per attribute (w = 10^6). Moreover, data is in BCNF and sorted.

Regarding factorized matrix operations, Reptile is compared against Lapack, which is a heavily optimized and widely used linear algebra library. The number of hierarchies d dictates the size

of the feature matrix X: exponential in the number of rows, and linear in the number of columns.

When using 7 hierarchies, Reptile is 5× faster by exploiting redundancies in the matrix and the range sum optimization. For right multiplication, Reptile is 1.6× faster by exploiting overlaps between vertically adjacent rows.

Then, the benefits of Reptile´s work-sharing multi-query optimizations are evaluated for computing the decomposed aggregates COUNT, COF, and TOTAL. The results are compared against LMFAO, which is the state-of-art factorised batch aggregation engine.

Reptile is over 4× faster than LMFAO, primarily due to optimizations based on independence between hierarchies.

In the case of Drill-Down Optimization, the evaluation includes static recomputation, dynamic exploitation of hierarchy independence, and caching of results. Moreover, Dynamic optimization is > 1.2× faster than static, and caching eliminates certain recomputation costs.

Finally, an evaluation of Reptile's end-to-end runtimes on real-world datasets: Absentee and COMPAS.

For both datasets, the initial complaint is that the overall COUNT is too high, and the models are trained using 20 EM iterations. It is shown that Reptile is over 6× faster than Matlab, 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.

Explanation Accuracy: Synthetic Data:

This section evaluates the accuracy of Reptile in identifying group-wise data errors using synthetic data. Synthetic data refers to artificially generated data that simulates characteristics of real-world data but is not derived from actual observations. It is created using mathematical models, statistical algorithms, or other data generation techniques.

The study focuses on how Reptile leverages complaints, hierarchical data, and multi-level models. Reptile´s creators designed the minimal experiment to evaluate how accurately Reptile can pick from the set of candidate drill-down groups. Therefore, a dataset with one dimension attribute was generated (i.e., one hierarchy) that had 100 unique values (and thus 100 groups), and one measure for computing aggregates. The number of rows in each group was drawn from a normal distribution N (100, 20), and each measure value is drawn from N (100, 20). In each experiment, we generate 1000 datasets and report the average accuracy of the top group.

Auxiliary data is additional information provided by domain experts that may have a potentially weak correlation with correct aggregate statistics. This correlation is simulated by generating auxiliary tables for each aggregate statistic, including COUNT, MEAN, and STD. The STD (standard deviation) is specifically used in the evaluation of the Raw condition. Each auxiliary table mirrors the original dataset's dimension attribute and includes a measure that is correlated (with a correlation coefficient, ρ, ranging between 0.6 and 1.0) with the respective aggregate statistic. The generation of correlated random variables in the auxiliary data follows a procedure proposed by Iman and Conover [23]. This auxiliary data is leveraged by Reptile in its analysis and recommendation processes to enhance the accuracy of identifying and resolving data errors.

Then, different error types are introduced, such as missing/duplicate records, and data drift to change COUNT and MEAN statistics. Other error types include Missing, Duplication, ↑ (increase), and ↓ (decrease), both individually and in combination. Afterwards, complaints are generated for COUNT, MEAN, and SUM based on the introduced errors.

Five different approaches are compared: Reptile, Outlier, Raw, Sensitivity, and Support.

- Reptile: Uses auxiliary data to repair the dataset and recommends the group that best resolves the complaint.
- Outlier: Ignores the complaint and returns the group with the most deviant statistics.
- Raw: A record-level approach based on Winsorization, clips measure values to resolve complaints.
- Sensitivity: Based on interventional deletions, recommends the group with the best resolution after deleting all rows.
- Support: A density-based approach that recommends the group with the largest COUNT.

When comparing the different approaches, Reptile consistently outperforms Raw, Sensitivity, and Support, leveraging auxiliary data even with weak correlation.

Outlier performs less accurately, especially in distinguishing between different types of errors.

Case study: COVID-19:

In this case study, the Reptile system was applied to COVID-19 data from the Johns Hopkins University Center for Systems Science and Engineering (JHU CSSE). The datasets included U.S. and global COVID-19 statistics, with hierarchies based on location (state, county, country) and time (day). The goal was to identify and rectify data errors reported in the GitHub issues of the dataset.

Ground truths for evaluation were obtained from confirmed and resolved data errors reported in GitHub issues between December 2, 2020, and January 27, 2021.

Corrupt datasets were generated for each reported issue, and a complaint was formulated for each issue by aggregating total statistics at a higher geographical level and specifying whether the result was too high or too low based on ground truth.

Most reported issues were related to missing data on specific days, causing underreporting. Other issues included backlogged reports, changes in reporting methodology, etc.

For each issue, a complaint was created by filtering data by the complaint day, aggregating total statistics at a higher level, and determining whether the result was too high or too low based on ground truth.

Reptile was compared with two baselines: Sensitivity and Support.

Sensitivity is based on interventional deletions, recommending the group that, after deleting all rows, best resolves the complaint.

Support is a density-based approach, recommending the group with the largest COUNT.

Reptile showed higher accuracy (70%) compared to baselines (6.6% for Sensitivity, 3.3% for Support).

An error analysis revealed that some issues were due to minor data drift across several weeks, and others were subtle issues smaller than the natural variation in the data.

Conclusion: Reptile demonstrated its effectiveness in identifying and resolving data errors in the COVID-19 dataset, outperforming other approaches in accuracy despite taking longer for model fitting

Case study: FIST:

In the FIST case study, the Reptile system was applied to Ethiopian farmer-reported drought data collected by the Columbia University Financial Instruments Sector Team (FIST). The dataset included hierarchies based on geography (Region, District, Village) and time (Year), along with a severity measure ranging from 1 (low severity) to 10 (high).

Three FIST team members participated in using the Reptile system to submit complaints based on their experience with the data.

Users were shown visualizations of annual Region-level statistics and could click on suspicious statistics to create a complaint.

Example complaints included issues related to mean and standard deviation statistics in specific regions and years: “the MEAN in Tigray 2009 should be much higher because I remember farmers argued about this year (P1)”.

Reptile correctly identified errors for 20 out of 22 complaints submitted by FIST team members.

Identified errors revealed issues such as confusion between planting and harvesting years, misremembered events, and reporting non-drought years as highly severe.

Conclusion: Reptile demonstrated its effectiveness in assisting domain experts in cleaning and interpreting complex datasets, providing valuable insights and automation for group-level inspection. Users appreciated the system's contribution to scalability and time-saving in the data cleaning process.

Conclusions:

Reptile is introduced as a powerful system for iterative error identification and repair in aggregation query results. Employing a factorized matrix representation and hierarchical structure optimizations, Reptile significantly outperforms alternative implementations, reducing runtimes by over 6×. It operates by recommending drill-down actions and estimating aggregate statistics for each group using a trained model. In practical applications, Reptile demonstrated its efficacy by identifying a majority of errors in COVID-19 data from Johns Hopkins University and successfully addressing complaints in a user study with the Columbia University Financial Instruments Sector Team, highlighting its value in real-world scenarios. The system's ability to integrate machine learning with visual analysis patterns contributes to its impact and potential for scalable and efficient data cleaning processes.