

Synthesizing Analytical SQL Queries from Computation Demonstration

This page hosts the artifact for our PLDI 2022 submission, [Synthesizing Analytical SQL Queries from Computation Demonstration \(draft\)](#).

A more readable google doc version of the Readme.txt is in the following link:

<https://docs.google.com/document/d/1gYMYn4YXqSkn6WrUL786IS5hKYn25NZxLzcBTDZkkTQ/edit?usp=sharing>

Materials

Our artifact contains the following materials:

- *Benchmarks* of Analytical tasks. See the Benchmarks section.
- *Implementation of the synthesis algorithm* (named SickLe, stored under the `artifact/sickle` directory) in Python3. This is our implementation used in the paper evaluation.
- *Evaluation log and analysis scripts* used in our paper evaluation.

Getting Started

Our experiments require anaconda installed. You can find a more detailed install tutorial on <https://www.anaconda.com/products/individual>.

After the installment, run the following commands in the anaconda prompt window to create a new environment with all required dependencies. Select “yes” to any prompts for proceeding. (This process may take a few minutes)

```
conda create -n sickle_artifact python=3.8 r r-essentials r-dplyr  
r-ggplot2 r-jsonlite pandas numpy tabulate
```

Then, you can activate the environment with the following command

```
conda activate sickle_artifact
```

After activation, you should be able to see your current environment has been changed to (`sickle_artifact`). If there is any error with the conda environment you created, you can deactivate the environment with `conda.bat deactivate` to quit the environment and re-do the steps above to create a new environment.

Overview

In the artifact, we included implementations of an analytical query synthesizer that uses enumerative synthesis algorithm and provenance abstraction for pruning the search space. In addition, we included the implementation of baseline synthesizers that use value abstraction, and type abstraction for running experiments. We also included 80 real-world benchmarks on solving analytical SQL problems with synthesizer. In the following, we first present benchmarks we used in our evaluation, then show instructions to reproduce our experiments described in the paper draft.

Benchmarks

We collected 80 real-world benchmarks on analytical tasks and stored them in `artifact/benchmarks`. Benchmarks numbered from 001 to 060 are collected from forum discussions and tutorials; benchmarks numbered from 101 to 120 are adapted from TCP-DS decision support benchmarks. Each benchmark is documented in Json format and contains the input table, the url of where the benchmark comes from (some do not have source urls), the correct query to synthesize (The correct program is used for checking the correctness of synthesis results and for generating a simulated demonstration for the task. See section 5 of the paper for more details about experiment setup), and a hard-coded configuration for aggregation operators and parameters specified to the synthesizer.

The expected program is stored as a list of json objects, with each object representing one step of operation. In our implementation, we have four major operators:

- `Group_sum`: an operator equivalent to group by aggregation in analytical SQL which takes group columns, aggregation function, and aggregated columns as parameters. The implementation allows aggregate on multiple columns at one step.
- `Group_mutate`: an operator equivalent to partition by aggregation in analytical SQL which takes partition columns, aggregation function, and aggregated column as parameters.
- `mutate_arithmetic`: an operator that takes a lambda function and columns that participate in the computation as parameters
- `Join`: an operator equivalent to join in analytical SQL which takes two input tables, a join predicate, and a boolean-type value indicating if the join is a left-outer join or inner join as parameters. In the implementation, we only allow joining two tables at one step and use a left-deep join tree.

Note: the projection operation is supported from the containment check, so there is not an exclusive operator for it.

In the parameter configuration, we stored parameters for the operations described above:

- `Aggr_func`: aggregation functions allowed for the `Group_sum` operator.
- `Mutate_func`: aggregation functions allowed for the `Group_mutate` operator.
- `Join_predicates`: user-specified predicates with the first tuple indicating the ids of join tables and the second tuple indicating the column ids of the join keys in the two tables.
- `Join_outer`: a boolean-type value indicating whether the join is a left-outer join or inner join.
- `Mutate_function`: arithmetic functions allowed for the `mutate_arithmetic` operator.

If the benchmark file does not contain a parameter-configuration, we will use a default configuration stored in `configuration.py` for running experiments. The default configuration contains all aggregation and arithmetic functions we support in this artifact.

Evaluation Instructions

In paper, we demonstrated experiments that allow us to evaluate the run time and program search space of sickle and compare the performance of provenance abstraction with type and value abstraction. In the following section, we will run the same experiments over the 80 benchmarks to reproduce the results shown in the paper.

Run the following command to start the experiment.

```
python artifact/sickle/eval_main.py
```

On average, each run of the evaluation script takes around 8 - 10 hours (for all 80 benchmarks on the three algorithms described in the paper), or longer depending on your environment. When there is an unexpected crash, you will have to restart the experiment from the beginning. However, the log of results before the crash will be preserved in `running_result.json` for you to check the results of the benchmarks that have been run.

The complete log of the experiment result will be under the output folder in `running_result.json`. Each line of the `running_result.json` shows the experiment result on each benchmark. In dictionary format, the resulting line for each benchmark contains the id of the benchmark, the time cost, the number of programs visited, the number of consistent results found, and an indicator of whether the time cost exceeds the required time limit.

We also stored logs for the experiments on each benchmark under `artifact/runtime_log`. Each log file contains more detailed information of the

experiment setup and results. The configuration number in the file name indicates which approach we used for synthesizing with 0 representing provenance abstraction, 1 representing value abstraction, and 2 representing type abstraction.

Remarks

- We also provide raw logs produced by our experiment for paper submission in the `artifact/output` directory (namely, `paper_running_rlt_pldi.json`). You can copy this data into the `log_result.json` file and follow the same process to draw the plot we included in the paper.
- You may get varied running results due to randomness designed in the experiment (a random seed is pre-set for you to keep consistent results over multiple runs). For each benchmark example, we generated the user example by randomly selecting rows and provenance information from the documented correct result. The preciseness and conciseness can result in different run-time cost and the number of programs searched.

Reproduce Evaluation Results

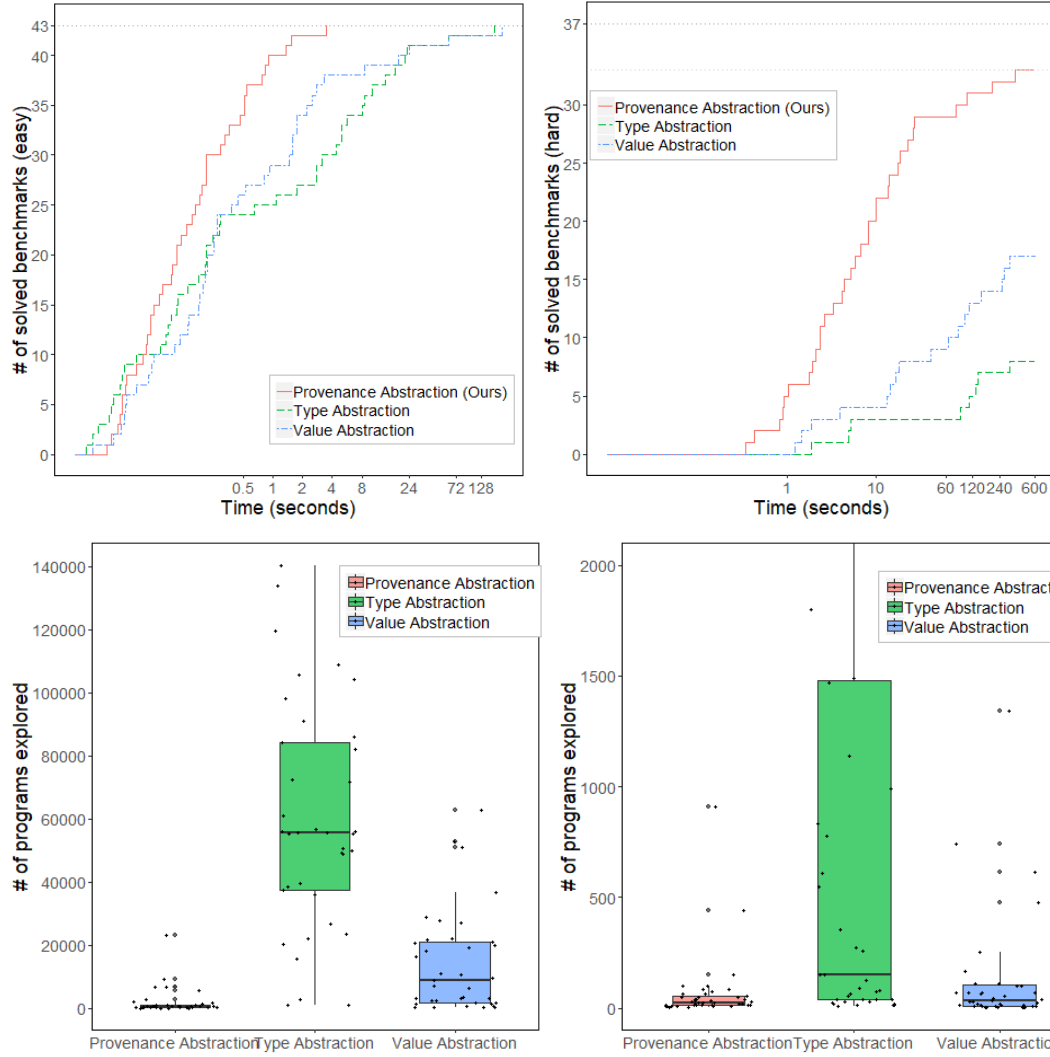
After obtaining experiment results, we can run the analysis script to obtain the evaluation results reported in the paper. Use the following command to run the analysis script:

```
Rscript ./artifact/sickle/plot_results.R ./artifact/output/running_result.json
```

Alternatively, you can run the following command to reproduce the plots shown in the paper

```
Rscript ./artifact/sickle/plot_results.R  
./artifact/output/paper_running_rlt_pldi.json
```

The experiment result plots used in the paper draft are shown as followings:



The generated plot is stored in the `artifact/output` directory as png files and should have the same shape as the plot shown in Figure.12 and Figure.13 in the paper. Note: the plot generated may not completely fit the plots shown above due to the random nature of the experiment setup.

Code Structure Explanation:

In this section, we explain the purpose of each script component for the synthesizer implementation and experiments in the directory `artifact/sickle`.

- `synthesizer.py` and `table_ast.py` contain the implementation of operators' sickle supports, the pruning rules, and an enumerative synthesis algorithm using the rules to prune search space.
- `table.py`, `table_cell.py`, and `table_cell_structureless.py` contain data structures for building annotated tables that store provenance information.

- `plot_results.R` and `eval_main.py` are used to produce experiment results and generate plots used in paper.

Extend Synthesizer:

To extend the synthesizer, you can support new operators by implementing Node class in `table_ast.py`. You will need to implement the following functions in each sub class you created:

- `Infer_domain`: a function that provides all possible parameters to enumerate onto the synthesizer.
- `Eval`: a function that runs the operator with the input table and stores the trace information. This function should return an annotated table (see `table.py` for more details) that contains the provenance trace for each cell in the resulting table.
- `Infer_trace`: a function that contains the implementation of a pruning rule for each table. This function should return an annotated table that represents an over-approximation of the partial program result. Refer to Figure.11 in the paper for pruning rules of the key operators we implemented in this artifact.

Please refer to the existing operator implementation for more details of creating a new operator.

Playing with the Synthesizer:

To use the synthesizer, you will need to create a json file containing the input table and the visual sketch similar to the following example and put it into the `sickle/source` folder.

The input file you created should contain the following components similar to which contained in the provided example:

- A list of input tables:

"input_data":

[[

```
{
  "city": "A", "enrolled": 1667, "quarter": 1, "group": "Youth", "population": 5668},
  {"city": "A", "enrolled": 1367, "quarter": 1, "group": "Adult", "population": 5668},
  {"city": "A", "enrolled": 256, "quarter": 2, "group": "Youth", "population": 5668},
  {"city": "A", "enrolled": 347, "quarter": 2, "group": "Adult", "population": 5668},
  {"city": "A", "enrolled": 148, "quarter": 3, "group": "Youth", "population": 5668},
  {"city": "A", "enrolled": 237, "quarter": 3, "group": "Adult", "population": 5668},
  {"city": "A", "enrolled": 556, "quarter": 4, "group": "Youth", "population": 5668},

```

```
{
  "city": "A", "enrolled": 432, "quarter": 4, "group": "Adult", "population": 5668},
  {"city": "B", "enrolled": 2378, "quarter": 1, "group": "Youth", "population": 10541},
  {"city": "B", "enrolled": 1200, "quarter": 1, "group": "Adult", "population": 10541},
  {"city": "B", "enrolled": 1373, "quarter": 2, "group": "Youth", "population": 10541},
  {"city": "B", "enrolled": 853, "quarter": 2, "group": "Adult", "population": 10541},
  {"city": "B", "enrolled": 246, "quarter": 3, "group": "Youth", "population": 10541},
  {"city": "B", "enrolled": 235, "quarter": 3, "group": "Adult", "population": 10541},
  {"city": "B", "enrolled": 768, "quarter": 4, "group": "Youth", "population": 10541},
  {"city": "B", "enrolled": 801, "quarter": 4, "group": "Adult", "population": 10541}
],
```

- A computation demonstration with “op” fields being computational operation or aggregation and “children” fields being the provenance input for the operation (Note: since we do not support a complete interface, any incorrect formats of input demonstration may result in failure to synthesize the correct result or an error)

```
"demonstration": [
  [{"o_a0"}, {"op": "lambda x, y: x / y * 100", "children": [{"op": "sum", "children": [{"o_b0", "o_b1", "o_e0"}]}]},
  [{"o_a7"}, {"op": "lambda x, y: x / y * 100", "children": [{"op": "sum", "children": [{"o_b0", "o_b1", "_UNK_", "o_b7", "o_e7"}]}]}
],
```

- A parameter configuration containing all the computational operations used in the demonstration that synthesizer should enumerate on.

```
"parameter_config": {
  "operators": ["group_sum", "group_mutate", "mutate_arithmetic", "join"],
  "aggr_func": [],
  "mutate_func": ["sum", "cumsum"],
  "join_predicates": [],
  "join_outer": [],
  "mutate_function": ["lambda x, y: x / y * 100"]
}
```

Note that the trace information is specified as cell coordinates where column number is specified with alphabets for clarification cleanness. For example, a coordinate “0_a0” means the cell is from table 0, in the first column, and in the first row. To omit some coordinates, you can insert a special “_UNK_” at any position of your trace list.

Use the following command to run the example. The command takes three parameters: the path from the root directory to the input file, the time limit (default 600 seconds), and the number of candidate solution to show (default 5)

```
python ./artifact/sickle/run_sickle.py ./artifact/sickle/source/example_1.json  
600 5
```

Use the following command to run the input file you created.

```
python ./artifact/sickle/run_sickle.py  
./artifact/sickle/source/<YOUR_INPUT_FILE>.json 600 5
```