DREAM final report

Faculty advisor: Eugene Wu
Julia Bell
Aug. 16 2023

Abstract

This paper explores an attempt to generate and retrieve an intervention matrix, a columnar representation of bit vectors which indicate if a certain attribute value is present in some tuple for a given relation. The intervention generation attempts to speed up existing intervention matrix generation in the query explanation engine FaDE (**Fa**st **D**eletion **E**valuation), where the intervention matrices are used as counterfactual data deletions and the matrix is read from disk at runtime. In this proposed version, space is minimized using a 64-bit encoding scheme, and the matrix is generated in DuckDB at runtime. While this method was developed to reduce throughput, initial tests did not evaluate faster than the existing method. However, more testing and development is needed to conclusively determine its value.

Introduction

This work augments FaDE, a query explanation engine that proposes explanations for anomalies by applying hypothetical deletion interventions over input data and evaluating effects on the anomaly. If the impact of some deletion intervention is high, it indicates that the deleted input tuples have a high level of influence on the anomalous data. As defined in FaDE, given Q(D) over the database D, identifying ΔD such that Q(D - ΔD) is closest to the expected result will determine ΔD is most influential to the anomaly[1]. The efficiency of finding the most influential result is determined by the number of interventions, and the speed of their evaluation.

Query explanation engines allow users to drill down into unexpected results, and explore explanations for them. This enables a higher level of user interactivity and analysis. This work addresses a specific part of the FaDE workflow: namely, the creation and retrieval of the intervention matrix. Within FaDE, an intervention matrix is created and a code generation module produces a C++ code snippet that computes the effect of deleting input tuples for a given query. When this C++ code is run, the hypothetical results of the query with interventions are calculated. As input to recalculate results, FaDE takes the intervention matrix and lineage artifacts, or structures containing provenance metadata[2]. Performing intervention evaluation quickly is essential to make interactive, real-time analysis possible: The current benchmark to achieve interactivity requires evaluating 1M interventions must be evaluated per second[3].

In the present FaDE implementation, the intervention matrix is materialized to disk and then retrieved at runtime. As writing and reading to disk are costly operations, improving the efficiency to evaluate intervention matrix has been a primary goal. Furthermore, the current intervention matrix is populated with randomized data that isn't meaningful for analysis. Creating meaningful intervention matrices that accurately represent the actual effect of deletion on the input data is another aim. Users should be able to indicate which attributes they wish to focus on, which will be made into a matrix. This work outlines the current approach, its performance against the existing implementation, and what these results mean for the next phase of the project.

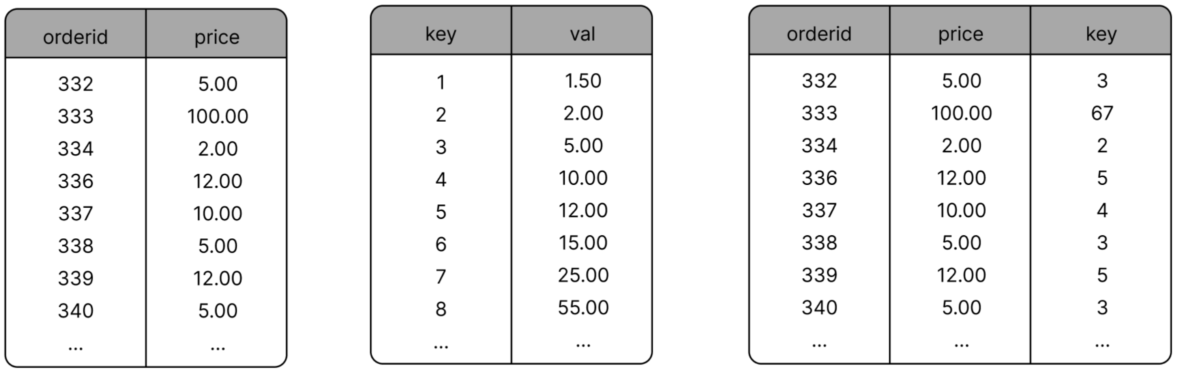
Encoding

As the goal for the intervention matrix was to do most of the intervention computation within the database, the specifics of the database system were considered in the encoding approach. The database used in this project is DuckDB, an embedded SQL database that allows databases to be created on a local machine or in memory. In this case, the database is accessed through the public Python and C++ APIs.

To represent the interventions, a dense encoding scheme was used. As the size of the intervention matrix grows with the number of values, bit-packing is used to save space. Given the system's use of 64-bit registers, 64-bit integers are used to encode 64 values per column.

The resulting intervention matrix is a columnar representation of binary annotations. For the *ith* column, the result is a bitwise vector where for each row j, [j][i] = 1 if code[i] = val and j.attribute = val, and [j][i] = 0 otherwise.

Figure 1



orderid	price
332	5.00
333	100.00
334	2.00
336	12.00
337	10.00
338	5.00
339	12.00
340	5.00
...	...

A. Original **orders** relation with attribute of interest, **price**.

key	val
1	1.50
2	2.00
3	5.00
4	10.00
5	12.00
6	15.00
7	25.00
8	55.00
...	...

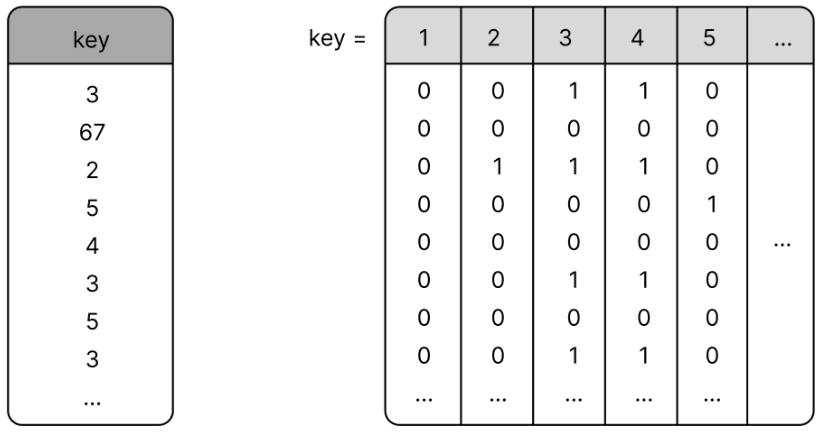
B. The **code** relation, ordered and numbered list of all discrete price values.

orderid	price	key
332	5.00	3
333	100.00	67
334	2.00	2
336	12.00	5
337	10.00	4
338	5.00	3
339	12.00	5
340	5.00	3
...	...	...

C. Join product of orders and code relations, yielding a **key** column with encoded values from price column.

This approach was developed as an alternative to using queries for bitpacking, which was shown to be inefficient (e.g. `SELECT a = "1" | a = "2" << 1 | a = 3 << 2 ...)`. Instead, bitshifting elements were incorporated as part of the intervention matrix retrieval query. Logically, each bit-length column is a single intervention. Physically, each column is a 64-bit integer, packing up to 64 interventions into one column. This requires bitshifting each intervention within a group of 64.

Figure 2



key
3
67
2
5
4
3
5
3
...

A. **key** column for price attribute

key =	1	2	3	4	5	...
	0	0	1	1	0	
	0	0	0	0	0	
	0	1	1	1	0	
	0	0	0	0	1	
	0	0	0	0	0	...
	0	0	1	1	0	
	0	0	0	0	0	
	0	0	1	1	0	
	...	...	...	...	...	

B. resulting intervention matrix for all values of price attribute

The pseudocode shows the bitshifting process, where the bitshift function is called once for every 64 unique values per attribute.

C/C++
```
void f(vector<int> &input, int col_num, vector<unsigned long>
&output) {
  int r = 64;

  for(int n = 0; n < input.size(); n++) {
    unsigned long b = 1ULL;
    if(*input > r*i && *input <= (r*i + r)) {
      int shift = r - (*input - (r * i));
      b <<= shift;
    } else {
      b = 0ULL;
    }
    output[n] = b;
    input_data++;
  }
}
```

As an example, if an intervention matrix is to be created on the price attribute in the orders table. These SQL queries will create a view, code, that resembles the view in Fig. 1B, and the JOIN product, matrix, like Fig. 1C.

```
CREATE VIEW code AS SELECT CAST(ROW_NUMBER() OVER() as INT) as
key, val FROM (SELECT DISTINCT price AS val FROM orders ORDER BY
val);

CREATE VIEW matrix AS SELECT code.key AS key FROM lineitem LEFT
JOIN v ON code.val = orders.price ORDER BY ROWID;
```

The underlying relations are created as views for space efficiency. To determine how many times the bitshift function must be called, the number of distinct values must be known, which requires querying the view. In this iteration, this was computed offline for efficiency.

```
int num_distinct = SELECT MAX(key) AS max FROM code;
```

The resulting figure will be the number of discrete values per key, the attribute of interest. Finding the number of 64-bit columns in the matrix can be found with simple division, such as:

```
int cols = (num_distinct / 64) + (num_distinct % 64 & 1);
```

This returns the number of columns of 64 values each, and is the basis for generating the correct query. For instance, 130 values would require 3 columns, and the resulting query will be: `SELECT udf(key, 0), udf(key, 1), udf(key, 2) from matrix;` The bitshifting is computed when the query is run, in a user-defined function (called `udf` here). Each user-defined function scans the entire input relation, which it takes as the parameter key. Therefore the number of columns also indicates the number of scans over the entire input relation, although since DuckDB handles the UDF execution, it may actually be called more times, with the input tuples broken into optimal groups determined by DuckDB.

Given the use of DuckDB, one consideration is if the query results should be materialized in memory, or return as a cursor. The cursor is more space efficient, and values can be iterated through with pointers much faster than the methods provided by DuckDB for finding values in the materialized result. The downside of the cursor is that the entire result is not materialized at once, rather results are materialized in “pages” of 2048 rows. A built-in function allows users to navigate through the “pages” of results. In the current implementation of FaDE, the results of JOIN queries must be fully materialized so the entire original row order of the result is accessible. However, other operators should be able to use the cursor result. FaDE templates can be modified to reflect this logic.

Implementation and testing

To create a prototype for testing the encoding and retrieval of intervention matrices, existing FaDE code was modified. Namely, additional parameters were added to accept user-submitted arguments for the attribute and table that should be used to create the intervention matrix. This work was solely focused on single-attribute interventions (e.g. STATE = “NY”), while future iterations of FaDE will include multi-attribute conjunctive interventions (e.g. STATE = “NY” AND GENDER = “F”).

Benchmarking is done using the Google Benchmark library. Scale factors (SF) of 0.001, 0.01, 0.1, and 1 reflect input size of 1500, 15000, 150000 and 1500000 respectively. It is assumed that the databases are initialized with the correct scale factor, and the proper query is constructed offline, in the method discussed above, and passed to the C++ code module as a parameter.

The previous FaDE system and the one with the updated code were tested on the same simple GROUP BY query. A single-column intervention matrix was constructed.

The following timings were measured for various scale factors using different methods of generating and retrieving the matrix. These timings are the time it took to run the generated C++ code:

SF	Read matrix from disk	Cursor result from DuckDB	Materialize result from DuckDB
1	143447550	559832167	236065139
0.1	14463447	37533821	23320982
0.01	1430832	8523554	5689721
0.001	142835	2597842	2065875

Timing in nanoseconds

These timings indicate that reading from disk is faster than retrieving the result from a DuckDB view. This was not the expected result of retrieving the matrix from DuckDB.

To try to understand this result, the code to fetch the matrix was broken down and measured to understand how long it took to execute each part.

These are the rough steps to connect to some DuckDB instance and query it, where the final step, `result→Fetch()` retrieves the results from the cursor so they can be read.

```C/C++
// Connect to database
DBConfig config;
DuckDB db("build/newdb/newdb0.001.db", &config);
Connection con(db);

// Register UDF
con.CreateVectorizedFunction("udf", {LogicalType::INTEGER,
LogicalType::INTEGER}, LogicalType::BIGINT, &udf);

// Query database
string q = "SELECT udf(key, 0) FROM matrix;";
auto result = con.Query(q);

// Fetch results
auto chunk = result->Fetch();
```

For a scale factor of 0.001, the steps are annotated in this code, and the timing for each is reflected in the table below:

Connect to database	1300523
Register UDF	1324312
Query database	2091188
Fetch results	2091203
Total time	2597842

Timing in nanoseconds

As this table shows, almost all of the time it takes to execute the query (including fetch other lineage artifacts and construct the new result) is taken up by connecting to the database and querying it. Connecting to the database takes over half the total time alone, and fetching the results takes a little more than a quarter of the total time. Registering the UDF and fetching the results from the cursor are largely negligible.

Conclusions

It’s not conclusive that fetching intervention matrices from DuckDB will improve the overall efficiency of FaDE. As these initial tests show, the cost of connecting to DuckDB may be prohibitive. However, it’s not clear that this is definitively the case, as it’s possible that there are unstudied benefits to this method, or situations where it would be more efficient than the original implementation. Namely, this method should be comprehensively tested against the original implementation with varying sizes of intervention matrices. It’s possible that as the size of the intervention matrix grows, the cost of reading it from disk will eventually outstrip the fixed cost of connecting to the DuckDB database. This relationship could be plotted in further study.

Although the current focus was on testing a single attribute, multiple attributes and more tables can be supported by allowing the user to pass in a dictionary parameter, e.g {orders: [price, status], regions: [state]} to create interventions on the order.price, order.status, and regions.state attributes. As mentioned above, supporting multi-attribute queries is a key objective of FaDE.

Additionally, other avenues of optimization and development on this project are in nascent or exploratory stages of development. This work only focused on a dense encoding scheme, although other analysis on a sparse encoding scheme may discover that the sparse representation leads to more efficient generation. There also may be further opportunities to improve the speed of evaluation through parallelization. As these avenues, and others, are explored fully, the question of how to generate intervention matrices most efficiently will become more understood.

---

[1]A. Yao, L. Flokas, E. Wu. “FaDE: Answering ”Why?” Made Fast.” In Progress 2023.

[2] F.Psallidas, E. Wu. “SMOKE: Fine-grained Lineage at Interactive Speed” 2018.

[3] E. Wu. “Interactive Explanatory Analyses for Big Data Business KPIs” 2023.