Unshackling Database Benchmarking from Synthetic Workloads

Parimarjan Negi*[1], Laurent Bindschaedler*[1], Mohammad Alizadeh[1], Tim Kraska[1],
Jyoti Leeka[2], Anja Gruenheid[2], Matteo Interlandi[2]
[1]*MIT,* [2]*Microsoft*
pnegi@mit.edu, bindscha@mit.edu, alizadeh@csail.mit.edu, kraska@mit.edu
{first-name}.{last-name}@microsoft.com

***Abstract*—Introducing new (learned) features into a DBMS requires considerable experimentation and benchmarking to avoid regressions in production (customer) workloads. Using standard benchmarks such as TPC-H and TCH-DS is common practice, but, unfortunately, these do not represent the complexity of real production workloads. To solve this problem, in this demo, we propose a technique that generates a synthetic dataset from query logs and metadata—without touching the original data. The keystone of our approach is to map the data generation as a SAT problem where constraints, such as runtime cardinalities, are extracted from query logs and metadata. We show that our approach can generate representative benchmarks mirroring the performance of the original data without trading off privacy. The demo will guide the attendees through the various steps involved in the data generation and testing process.**

***Index Terms*—synthetic data generation, benchmarking, SAT.**

I. Introduction

Large-scale analytics engines, such as Spark, SCOPE, Synapse, BigQuery, Redshift, and Snowflake, have become a core dependency for modern data-driven enterprises to derive business insights. At Microsoft, for example, the SCOPE big data analytics query engine in Cosmos is deployed over hundreds of thousands of machines, every day executing hundreds of thousands of jobs. These jobs are written by thousands of developers, process several exabytes of data, occupy millions of containers, and consume several petabytes of I/O [1].

Implementing new DBMS features, such as changes to storage layout [2], [3], addition of new rules to the query optimizer [4], or learned systems [5], [6], requires significant benchmarking to avoid performance regressions that can impact thousands of users. The current approach to benchmarking relies on synthetic workloads, such as TPC-H [7] or TPC-DS [8]. However, these benchmarks do not cover the vast diversity of real-world production workloads [9]. Real-world workloads often differ in many dimensions, such as the types of operators, the complexity of filters, or query templates. Fig. 1 illustrates these differences by comparing the selectivity of continuous filters (e.g., range predicates on a column) *vs* discrete filters (such as equality predicates) on several known DBMS benchmarks such TPC-H [7], TPC-DS [8], Cardinality Estimation Benchmark (CEB) [10], and Join Order Benchmark [11], along with real-world workloads from a SCOPE cluster. The wide diversity of real-world workloads suggests that evaluating new features only on synthetic benchmarks may overfit the few covered cases. In turn, this approach could indicate good performance on paper, but suffer considerable performance regression when deployed in production.

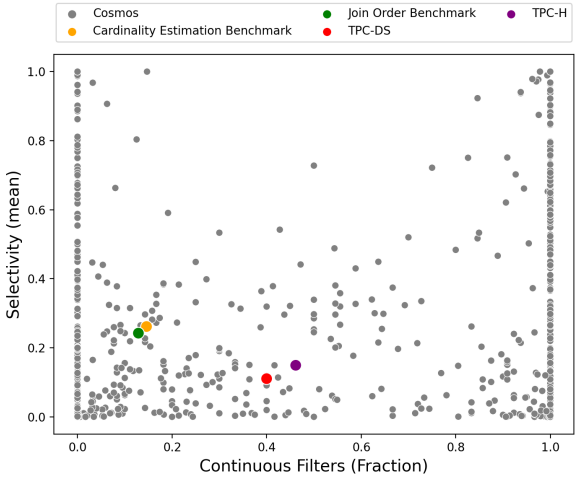

Fig. 1: Fraction of continuous operators in queries vs average selectivity. Each point represents a collection of inputs in Cosmos (gray), or a standard DBMS workload (colored).

DBMS workloads can be split into three main components: the *data*, the *queries*, and *metadata* (e.g., cardinalities generated by executing the queries on the data, or the schema information of the database). Among the three components, data is the most challenging to acquire for benchmarking purposes since it is generally not feasible to access real-world customer data due legal or privacy concerns. Conversely, accessing anonymized query logs and metadata is more straightforward: in fact, several cloud systems already store this information for offline historical and post-mortem analysis [1].

This demo presents a first step towards automatically synthesizing databases from workload logs containing anonymized queries and metadata information such as runtime cardinalities and schema information. The resulting data can then be used along with the queries as a *synthetic bench-*

* Equal Contribution.

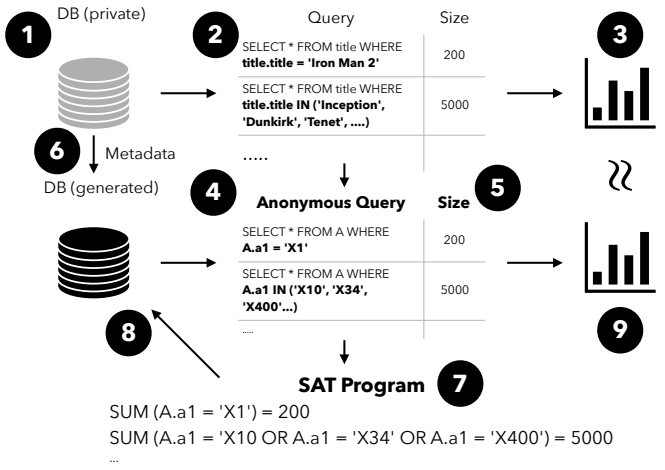

Fig. 2: Overview of our approach.

mark mirroring the characteristics of the original workload. Other approaches have been proposed to solve a similar problem [12], [13]. However, these approaches: (1) use a distribution-based approach which breaks when the distribution from query logs does not match the one in the data, or (2) require accessing the original data (e.g., SAM [12] learns the correlation across tables by executing an outer join). The keystone of our approach is to look both at queries and metadata information, and consider them as constraints. We then map the data generation to a SAT problem and use off-the-shelf constraint solvers [14] to generate a possible solution. To the best of our knowledge, our approach is the first to show that mapping the data generation to a SAT problem allows generating datasets with similar latency as the real database on the given queries. In the demo, we will guide the audience through our approach using two open-source databases.

In the rest of this paper, we show the different steps required to generate new synthetic benchmarks: (1) parsing the input query logs and metadata into an intermediate representation (IR); (2) generating a SAT program from the IR; (3) solving and post-processing to create the synthetic database; finally, (4) we compare the generated database on the input workloads against the original one.

II. System Description

Fig. 2 provides an overview of our synthetic benchmark generation. We start with a customer workload containing a private dataset (❶) and a set of queries (❷). The customers submit the queries to a cloud data warehouse system such as SCOPE. The output consists of the performance of these queries (❸), and a set of anonymized queries (❹), with runtime statistics (such as cardinalities of intermediate results) (❺). Our system takes as input the anonymized queries, runtime statistics, and schema metadata (❻), and maps them to a SAT program (❼). The solution to the program is a synthetic dataset (❽) that can subsequently be used to run the anonymized queries. We define the pair of synthetically generated data and the anonymized queries as a *synthetic benchmark*. We aim for this synthetic benchmark to be representative of the original workload, i.e., its execution performance (❾) is similar to the original (❸). Next, we formally define the problem.

A. Problem Statement

At a high-level, our approach consists of translating the filter predicates of SQL queries and the corresponding runtime statistics into constraints, specifically a Boolean satisfiability problem (SAT) that we then solve using an off-the-shelf solver. The ultimate goal of this approach is to assign one value per element in a series of matrices that represent the tables of the original database. More formally, assuming that the database consists of n tables, we define one representative matrix M_T per table T with each element $M_{T\,i;j}$ corresponding to a variable whose domain ranges over all possible values of column j in the original table and `NULL`. Since different queries may cause the domains of the same variable to overlap, we opt to represent each variable $M_{T\,i;j}$ as a series of Boolean variables $M_{T\,i;j;v}$ for all possible v values in the domain. Only one $M_{T\,i;j;v}$ can be `true` for all v, allowing us to easily decode the corresponding value for $M_{T\,i;j}$.

Each filter constraint limits the possible values of each variable and any assignment to the variables that satisfies all constraints is a solution to the problem. Given any valid solution, we can reconstruct the data in the database such that the SQL queries will lead to the same selectivity and execute with a similar performance profile.

B. Pre-processing

Pre-processing involves several steps that collect necessary data to build the appropriate models. Starting from an anonymized query log, we derive the columns and filters along with a series of *column-based* partitions. Fig. 3 illustrates the different pre-processing steps and results. Note that the original database is shown for illustrative purposes and is not assumed to be accessible for pre-processing. We provide scripts to extract information from query logs generated by different DBMS (e.g., Postgres and SCOPE).

Derived columns. We identify all queried columns and their types from the query log, thereby deriving a subset of the original database schema.

Derived filters. We extract from the query logs the filter conditions as well as the cardinalities from the runtime statistics.

Partitions. We partition the filters by their covered columns, resulting in disjoint sets of queries and SAT problems.

Downsampling. Given a large table and related domains for its variables, the SAT problem can easily reach millions to billions of variables. The off-the-shelf solver we are using [14] works reasonably well for SAT problems with at most `100` million variables. Therefore, we achieve scalability by downsampling each table to a lower number of rows, adjusting the selectivity of each query in the process. We address lost filters whose selectivity downsamples to zero during post-processing (§II-D), by adding the corresponding values uniformly at random after upsampling. As evaluated in Section III, we find that, in most

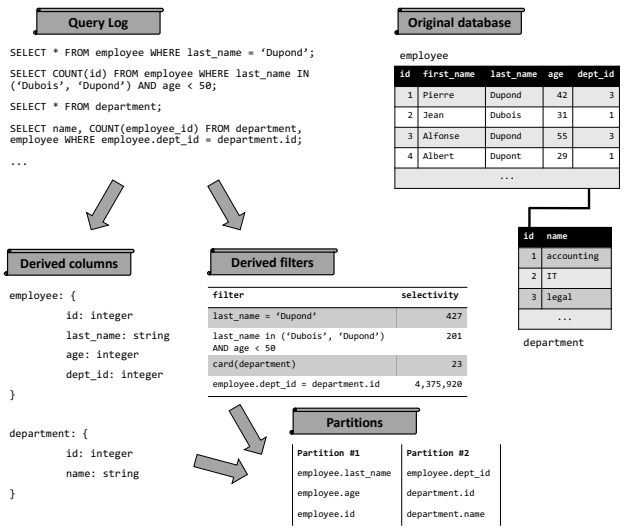

Fig. 3: An example of pre-processing.

cases, our approach does not significantly affect the results due to the low performance impact of these filters.

C. SAT Programming

After pre-processing, we obtain the derived columns, filters, and corresponding partitions. We can then proceed with modeling the problem as a SAT program. We illustrate the modeling process in Fig. 4, for the example introduced above.

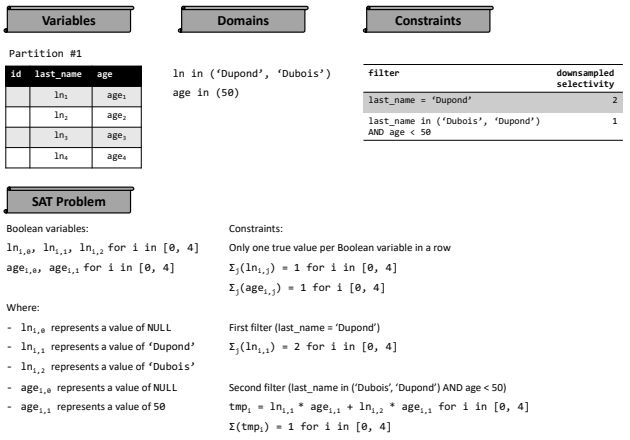

Fig. 4: Modeling the example of Fig. 3 as a SAT program.

We pick a downsampled cardinality for each table in the partition and assign one variable per row and column. We next select only those downsampled filters (constraints) for the current partition, and collect the literals to compute the domain of each column. Note that we assume that the domain of a column is equal to the set of all queried values, i.e., a value which is not queried is ignored and will not be represented in the SAT program (or the final solution). This assumption is reasonable for our goal of matching the performance of the given set of queries only.

We encode the SAT program as follows. For each column and for each row in each table, we assign a series of Boolean variables corresponding to each value in the column's domain and the special Unspecified value. For example, in Fig. 4, the `last_name` column has 3 possible values in its domain (Unspecified, "Dupond", and "Dubois"). For each row i, we assign three Boolean variables ($ln_{i;0}$, $ln_{i;1}$ and $ln_{i;2}$, corresponding respectively to Unspecified, "Dupond" and "Dubois". We add a constraint for each row requiring that only one Boolean variable per column is set to `1` (`true`).

Having encoded all variables, we now process each filter and add the corresponding constraints to the SAT program. These constraints encode the selectivity of each query, forcing any solution to the program to match the required selectivity. In our example, the first filter (`last_name =` "Dupond") is encoded as a sum constraint, i.e., two rows in the table must have $ln_{i;1}$ (corresponding to "Dupond") set to `true`. Finally, we encode the second filter (`last_name in` ("Dubois", "Dupond") `AND age < 50`) using temporary variables to correlate the values of the two columns spanned by the filter. These temporary variables enforce that the variables satisfying the filter in each column appear in the same row to ensure that the selectivity of the filter is correct. Each temporary variable simply requires that $ln_{i;1}$ ("Dupond") and $age_{i;1}$ (50) or $ln_{i;2}$ ("Dubois") and $age_{i;1}$ (50) appear twice in the table.

The resulting SAT program can be solved using an off-the-shelf solver. Any solution to the program corresponds to an assignment that satisfies all constraints.

D. Post-processing

Post-processing consists of multiple steps that reconstruct a database from the solutions of each model. Here, we also utilize the schema metadata to build the final tables.

Combining solutions. We combine the different solutions to each model by creating the corresponding tables and placing the values in each column accordingly. Since each solution is assumed to be independent from others, this process simply involves appending the values to the appropriate column.

Upsampling. The resulting database is representative of the original database with the exception that its size is smaller due to downsampling. Therefore, we upsample the database by the same factor used for downsampling.

Filling in unspecified values. Recall, if the filter conditions do not cover the full table, the reconstruction will be partially unspecified. These unspecified rows are filled with:

1) Lost Filters: Due to downsampling, some filters with very low cardinalities were lost. These are filled in uniformly among the unspecified values in this step.
2) `null` values: The metadata information would give us the fraction of the column with `null` values—a fraction of the reconstructed column is filled in with `null`.
3) The remaining unspecified values are filled with random values sampled from the domain of the column; here we can use the statistics about column size to choose appropriate length values. This is necessary to get