Query Aware Synthetic Data Generation

Zoey Sun
Alvin Cheung, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-124
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-124.html

May 12, 2023

Acknowledgement

I express my profound gratitude to Professor Alvin Cheung and Xiaoxuan Liu for their invaluable assistance in my research endeavors. I am truly grateful for all the research insights, comments, and advice that they've provided me along the journey.

Query Aware Synthetic Data Generation

Mengzhu Sun

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Alvin Cheung
Research Advisor

5/11/2023

Date:

* * * * * * *

Aditya Parameswaran
Second Reader

Date: 5/11/2023

Query Aware Synthetic Data Generation

Mengzhu Sun
University of California, Berkeley
zoey_1124@berkeley.edu

ABSTRACT

Evaluating query workload on relational database is an essential task for many developers and researchers, but it is challenging to acquire relational data due to data privacy and confidentiality reasons. Query-aware synthetic data generation for database management system (DBMS) becomes crucial for benchmark testing. In order to ensure data fidelity, the synthetic data has to conform query cardinality constraints as well as properties of the database schema. Unfortunately, prior work for data generation either made simple assumptions about queries and database schema or fail to scale with large query workloads.

In this paper, we propose ezGen, a synthetic data generator for web application frameworks. ezGen decomposes complicates queries, especially subqueries, into cardinality constraints as data generator's input, then generating data using a probability approximation model. ezGen leverages a heuristic rule-based method to translate and decouple query-based cardinality into attribute-based cardinality. In addition, different from prior work, we aim to generate synthetic data for real-world database-backed web application testing by exploiting integrity data constraints extracted from application source code to further ensure the generated data fidelity.

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zoey1124/Touchstone_dev.

1 INTRODUCTION

Data generation in database management systems has been extensively studied in recent years [6, 11, 12, 14, 16, 17, 19], driven by the growing need to test database application performance. Synthetic data generation has emerged as a powerful technique for addressing privacy concerns and ensuring data confidentiality when evaluating database-backed applications. Query-aware synthetic data generation represents a particularly promising approach, as it enables the generation of synthetic data that not only preserves the statistical properties of the original data, but also incorporates the semantics of the queries that will be applied to the database. In this paper, we investigate popular prior approaches for query-aware synthetic data generation methods, and propose ezGen, a synthetic data generator tool that handles complex queries found in real-world database-backed applications and scale linearly with the size of generated data. We leverage data generation method from an approximation probabilistic model to capture the data distribution and intra-table dependencies from various queries.

The conventional approach for generating query-aware synthetic data for databases involved building a symbolic database under constraint followed by utilizing a constraint solver during the data instantiation phase. [6, 7, 11, 16]. For example, QAGen [7] focuses on using symbolic database targeting at a specific fixed query, and it is the motivation for many subsequent work [6, 11, 16]. MyBenchmark [16] leverages QAGen as a black-box processing each query and builds a bipartite graph to integrate symbolic database. Then the problem of instantiating symbolic data is reduced to a liner programming (LP) problem, and MyBenchmark uses a LP solver to find a solution. Because QAGen and MyBenchmark fail to produce a single database satisfying multiple query constraints, DataSynth [6] proposes to use a set of equations with variables to express cardinality constraints from all queries, and resolve the variable values with a LP solver to output one final database. However, LP-solver-based methods in general are subject to scalability limitations as the number of variables in the linear programming problem increase exponentially with the number of queries and generating data size. On the other hand, SAM [19] builds a supervised machine learning model from an existing database query workload, learning the cardinality constraints for data distribution. Nevertheless, developers might not have the access to original database workload, and learning data distribution from an existing database facing the risk of leaking data privacy and confidentiality. The state-of-art work for data generation problem is Touchstone [14], which adopts a new method based on a random column value generator using constraint chains. It is the most practical one to use because of its linear scalability with respect to the number of queries and number of generated records, the ability to generating data in parallel on different machines, as well as the austere memory consumption.

Unfortunately, Touchstone [14] fails to generate synthetic data for real-word database-backed applications for several reasons. While being able to handle testing benchmarks like TPC-H [5], real-world applications break some assumptions that TPC-H workloads hold like more complicated query logic and data integrity constraints extracted from web applications. This is because web applications will introduce more logic in source code when inserting a record into database [13, 15]. For example, application source code can require certain attribute data to follow special patterns or remain a unique value by checking the data record before inserting it into the database. In addition, a user has to manually analyze each query into query execution trees to use Touchstone, which makes it impractical for using it with large-scaled query workloads with hundreds or thousands of queries. Furthermore, Touchstone fails to handle complicated query logic, including query with complicated filter constraints, set operations, and subqueries.

To overcome the challenges mentioned above, here we present ezGen, a query-aware data generator that can handle thousands of complicated queries for web application databases. ezGen addresses the problem of data generator scalability with query workload and incorporates data integrity constraints from web application frameworks. In this paper, we claim the following contributions:

- **Query analyzer**. We introduce an innovative way to analyze query and extract information that related for cardinality constraints for query-aware data generation. We extract

basic operations in a query like filter, primary key join, and foreign key join to cardinality constraints that will be feed into the data generator. This allows users to directly throws queries they obtained to ezGen without any pre-processing steps, making ezGen easy to use.

- **Complicated query logic handling**. ezGen provide a set of case-by-case rule-based algorithm to handle complicated query logic including nested filter logic, set operations, and subqueries. The query analyzer eventually translates complicated queries into a series of cardinality constraints representing basic operations, i.e., filter and join.
- **Integrity data constraint handling**. ezGen allows user to specify data integrity constraints that are extracted from web application source code, including specifying attribute values to all be unique, not null, or from a certain predefined fixed value pool.

2 BACKGROUND

2.1 Problem Statement

In this work, we consider the input to our tool as a list of SQL queries $Q = \{Q_1, ..., Q_n\}$ and a database schema H describing information for tables $T = \{T1, ..., T_k\}$. Every query $Q_i \in Q$ is assigned a selectivity value to represent its cardinality requirement, as specified in the subsequent section. For each table $T_i \in H$, there is a set of data constraints on each attribute, specifying data type, value range, null percentage, average data length, as well as table size. The schema input H also provides relations between represented by table primary key and foreign key(s). In contrast to prior work [17, 19], we assume that we have no access to the original data in the database for user data privacy reason. It is reasonable to assume that each query Q_i may either contain pre-defined concrete values or undefined variables parameters denoted by P_j, as actions initiated on the application front-end (e.g., web page) are often mapped into queries, and the parameters in queries are instantiated with customized values based on user input.

Our goal is to populate all tables T such that queries return reasonable results, and the parameterized variables in queries are initiated. Under ideal cases, when executing large-scale testing queries, we expect query output to be non-zero for every query, such that the returned data records can well represent query performance for database testing and evaluation. However, in reality, some queries are naturally conflicting and cannot be satisfied at the same time. For example, if query Q_1 contains `WHERE users.age > 10` and Q_2 contains `WHERE users.age < 10`, it is impossible to satisfy $|Q_1| > 0$ and $|Q_2| > 0$, where $|Q_i|$ denotes the result cardinality of query Q_i. Therefore, given Q and H as input, ezGen will generate synthetic data to populate all $T_i \in T$ and instantiated all parameters P. The generated database should satisfy $|Q_i| > 0$ for as many Q_i as possible.

2.2 Approximation with Probabilistic Model

The biggest challenge for generating data to satisfy all cardinality constraints implied by queries is capturing the probabilistic distribution over multiple attributes on the filter nodes and join nodes in the query execution plan [10]. The query result cardinality is decided by such joint distribution on each node in the query execution tree. Notice that most of the database workload evaluation involves tuning the scale factor and changing the cardinality of generated relations for scalability test purpose. With the varying table size, we will use selectivity instead of cardinality to capture the result query size. Throughout this paper, we define selectivity on execution operation level instead of query level. For a given query Q, we first decompose Q into an execution tree with basic operations: filter on a set of relation attributes and foreign-key join operations. For filter operation(s) on table T_i, filter operation selectivity is defined with

$$S_\sigma = \frac{|\sigma(T_i)|}{|T_i|}$$

For a join operation on primary key attribute t of table T and foreign key s from table S, selectivity is defined as

$$S_\bowtie = \frac{|T_1 \bowtie_{(T_1.a=T_2.b)} T_2|}{|T_2|}$$

The selectivity definition is closely connected with data generation methods. Here for join selectivity, the denominator only involved foreign-key table T_2 because the foreign key attribute data will be generated after the primary-key data. The data for each table according to their partial order on primary-key/foreign key relationship. We make the assumption that the filter operations are on non-key attributes and join operations are on key attributes like many other related works [6, 10, 11, 14].

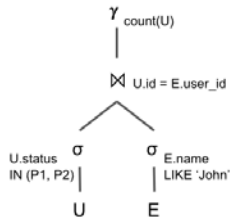
Selectivity value S should be provided by users. Users can specify a different selectivity value S for each execution node of each query or use a default selectivity value for all queries (e.g. set $S = 0.5$ for all Q). Notice that with thousands of queries, there will be thousands of nodes in query execution trees, and some of the query cardinality requirements are naturally conflicted as mentioned in section 2.1. For a pair of conflicting queries, when the final cardinality of one goes down, the other goes up. Observing the reversed relationship for conflicting queries, while MyBenchmark [16] choose to come up with multiple database instances satisfy each subgroup, we choose to consider all queries as a whole group. If there are any specific non-conflicting query subset that matters, user can also pull out the query subset and feed data generator. Also if we don't have access to original data, it is impossible to leverage "explain query" feature in database to see and calculate the selectivity for each node in the execution plan as mentioned in [14].

2.3 Cardinality Constraint

Cardinality constraints captures the expected outcome of a query executed over a database. It is crucial to translate queries into a set of formally defined cardinality constraints with expressiveness for data generator [6]. ezGen extract cardinality constraints form queries and adopts the similar notation from [14]. Let $A, B \in T$, for cardinality constraints extracted from all queries (i.e., $\forall c_i \in C$), we assume each c_i belongs to one of the following three categories:

(1) **Filter constraint** contains table information, attribute information, filter expression with parameters or values, and selectivity, denoted by $C^\sigma = (A, \sigma_{\text{attr op } P}, S)$. Notice that the filter expression could contain one or more filter operations, connected by logical operators `AND`/`OR`.

Figure 1: Execution plan for the basic query in listing 1.



(2) **Primary key join constraint** contains primary key table and join condition for an attribute from a table, denoted by $C_P^{\bowtie} = (A, \bowtie_{A.a=B.b})$.
(3) **Foreign key join constraint** contains foreign key table, join condition, and selectivity, denoted by $C_F^{\bowtie} = (B, \bowtie_{A.a=B.b}, S)$.

Let us illustrate with the query in listing 1. The corresponding query execution plan is shown in figure 1, which contains two filter conditions on table users and table emails respectively, and a join operation on both of the tables. We can extract 4 cardinality constraints as the following:

- c_1 = (users, users.type IN (P1, P2), 0.5)
- c_2 = (users, users JOIN emails ON id = user_id)
- c_3 = (emails, emails.sender = 'John', 0.5)
- c_4 = (emails, users JOIN emails ON id = user_id, 0.5)

Listing 1: An example query extracted from Redmine

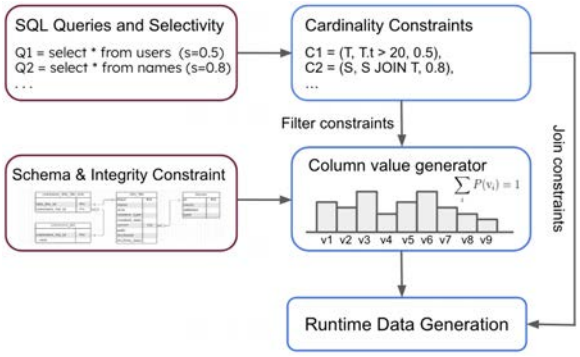
```
SELECT count(users.*) FROM users
INNER JOIN emails ON emails.user_id = user.id
WHERE users.type IN (P1, P2) and emails.sender LIKE 'John
    ';
```

Extracting cardinality constraints from queries allows us to express the information we need from "query-centric" to "attribute-centric". Then, we can group all cardinality constraint by each table attribute and build a column data generator based on that. The attribute-based generator adjust its value distribution for generated data in order to meet the expectation requirements from cardinality constraints, which will be explained in section 2.4. Therefore, it is important to separate constraints for each table attribute from the queries. For instance, there might be hundreds of queries targeting at the same attribute `users.age` in Redmine, with each constraints requiring for different values and ranges. By using our cardinality constraint, we can gather all relevant constraints on `users.age` attribute, and adjust the generator value distribution accordingly.

2.4 Adjusting Column Value Generator

The overall data generation process is shown in 2. The column value generator receives inputs in the form of cardinality constraints with selectivity values. For a non-key attribute, we can think of its value distribution as a histogram. Each value v_i is assigned with a probability $P(v_i)$ such that $\sum_i P(v_i) = 1$. With the filter cardinality constraints we defined above, the column value generator can adjust the distribution for each attribute to meet the selectivity expectation [14]. This is achieved by minimizing a global error, which is calculated as the difference between expected and actual

Figure 2: Overview of Data Generation Process



cardinality outputs for a given query. In other words, the column value generator processes all filter cardinality constraints C^{σ} and data type information for each attribute form schema as inputs, and returns a column value generator with the optimal value distribution to serve for run-time data generation. Then, during the run-time data generation, each data record is generated row by row, and the join cardinality constraints $C^{\bowtie}$ are checked to meet the expectation of join distribution.

2.5 Subqueries and Set Operations

While it is straightforward to extract cardinality constraints from basic queries by analyze operations like filter and join, the process becomes more intricate when dealing with complicated queries that incorporate set operations and subqueries. A subquery is a query that nested in a main query after clauses including `FROM`, `JOIN`, `WHERE`, etc. During query processing, the results of subquery is executed first, and the results are used by the outer query to determine the final result set. Hence, one cannot directly analyze cardinality constraints from such queries to ensure a non-zero results from the outer query. On the other hand, SQL set operations contains more than one independent queries, with a set operation from `UNION`, `INTERSECT`, and `EXCEPT`. We introduce a rule-based methods to parse and translate complicated logic in these queries into equivalent cardinality constraints in Section 3.

2.6 Web Framework and Data Integrity Constraint

In this work, we broaden the focus of query workload from the TPC-H industry benchmark to real-world database-backed object-relational mapping (ORM) applications [15] like Django [1], Ruby on Rails [4], and Hibernate [2]. ORM applications adopt a Model-View-Controller (MVC) logic pattern [18], with the Model part responsible for defining each class object to a table and maintaining the mapping between them. For example, for the two tables (i.e., users and emails) shown up in SQL query from listing 1, there will be two corresponding classes in application model. Inserting a record into the table users is reflected through `user = new User()` inside model layer.

The utilization of database in ORM web frameworks has been identified as a potential cause for the introduction of additional data

Table 1: Data Integrity Constraints with Code Examples

Integrity Data Constraint	Code Example
Unique	validate_uniqueness_of: *attr*
Not Null	validates_presence_of: *attr*
Foreign Key	has_many; belong_to
Inclusive	validates_inclusion_of: *attr* in value_list

integrity constraints via application source code, as stated by previous research [13, 15]. These data integrity constraints may include not are not limited to *not null*, *foreign key*, *unique*, and *inclusive*, which is listed in table 4. For example, in Ruby on Rails[4], *unique* constraint is only defined in application Model level, but is not enforced in database level. Due to the presence of two abstraction levels between application control and database management, it becomes challenging to maintain data quality that satisfies both levels of data constraints, and noncompliance with these constraints may result in severe consequences such as application crash or data corruption [8, 13]. Therefore, our work aims to enhance data fidelity from synthetic data generation by incorporating data integrity constraints extracted by tools from [13, 15].

3 QUERY ANALYZER

This section introduces the query analyzer for ezGen, which takes queries and output corresponding cardinality constraints we defined in section 2.3, so that the cardinality constraints can be used for data generator. One objective of the query analyzer is to decompose the cardinality constraints from queries into constraints on table attributes, incorporating selectivity in the process. By incorporating filter and join cardinality constraints, the data generator can be constructed initially with value distributions for all non-key attributes. This construction approach allows join cardinality constraints to be utilized at runtime to satisfy equality constraints over the join when generating data. Furthermore, the query analyzer assigns a selectivity value to each cardinality constraint. Recall that the selectivity of each node in the query execution tree represents the output/input size ratio. The selectivity value assigned to each cardinality constraint dominates the probability distribution for the data generator.

The functioning of the query analyzer involves an initial step of determining if the input query features set operations or subqueries (as discussed in section 4), followed by the application of specific rules for each case. Different rules are subsequently applied based on the type of operation identified. Further evidence is provided to demonstrate that the fulfillment of the extracted cardinality constraints can approximately meet the desired outcome of a populated database in the optimal way.

3.1 Query Analyzing without Nested Structure

Let us start with analyzing simple queries. Here, we use the word "simple query" to refer a standalone query without nested statement or set operation as defined in 3.1. The approach for analyzing simple queries involves constructing a hierarchy of Java classes for each query, followed by a visitor pattern that traverses through all class objects in the hierarchy to extract constraints on table attributes after SQL keywords like `WHERE`, `JOIN`, `HAVING`, etc. Notice that for aggregation queries, the pattern visitor only focuses on `GROUPBY` and `HAVING` clauses.

Definition 3.1 (Simple Query)**.** A query is called a*simple query* if it is a plain select query without set operations or any nested query structures (no subqueries).

For simple queries, ezGen query analyzer does not look into aggregation and projection because these operations will not affect basic row selections. The query analyzer also disregards filter constraints on primary-key attribute(s). This is because most of web applications default to use an auto-incrementing integer value as relation primary key (e.g., table_Id) for simplicity, immutability, and uniqueness purpose [9]. When ezGen generates primary key attribute, it also assigns an auto-incrementing integer to each new record. Therefore, a valid filter operation on primary key attribute will always return a unique record, and we can only focus on the join cardinality constraints on primary-key attributes.

3.2 Complex Filter Expression

Although simple queries contains no subquery, they could contain complex filter expressions like using nested parenthesis with AND/OR **binary operator** to connect a series of filter operations. For example, in listing 2, consider the `WHERE` expression from a query involving only a table T. We use lowercase letter (i.e, a - g) in this expression represents a basic filter operation like users.age > 20 on a table T, and the equivalent math expression shows its nested parenthesis and/or logic. ezGen proposes a precise way to capture the complex filter logic for calculating probability distribution of the column data generators and decide whether a generated record satisfy the filter logic.

3.2.1 Get probability of a filter expression. Let E represents the overall filter expressions which can followed by SQL keywords `WHERE` or `HAVING`, and let a lowercase letter represent a basic filter operation with a general form "attribute *op* value" (e.g., age > 20), where we call the *op* in the basic filter operation a **boolean logical operator** (e.g., "=", "<", etc.). Given each basic filter operation is satisfied with probability x (e.g., P (age > 20) = 0.5), we can calculate the overall probability $P(E)$ with algorithm 1. For example, let E_1 be the filter expression in listing 2 and each basic operation is satisfied with probability x, then:

$$P(E_1) = (1 - x^3) \times (1 - x^4)$$

Algorithm 1 cannot fully capture the precise probability model on each attribute since we don't have information about whether each pair of columns are mutually exclusive or not. However, it is a great approximation as the value of $P(E)$ is guaranteed between 0 and 1. This can easily be shown through induction: for basic case where E only has one filter operation, $P(E) = x \in (0, 1)$. With our induction hypothesis be $P(E_{\text{left}}), P(E_{\text{right}}) \in (0, 1)$, we can get the inductive step that $P(E)$ is always a valid probability value between 0 and 1. Using algorithm 1, we can get more precise probability for filter expressions.

Listing 2: An example query with complicated filter logics

```
1 -- filter clause from an example query
```

```
WHERE (users.name <> 'John' or (users.age > 20 and users.
      is_valid = TRUE)) and ((users.age < 20 and users.
      is_valid = FALSE) or (users.bit > 100 and users.name
       = 'Bob'))

-- extract filter logic
WHERE (a or (b and c)) and ((d and e) or (f and g))
```

Algorithm 1 Get probability of a filter expression

Input: Filter expression E consists of E_{left}, op, and E_{right};
The probability for each basic filter operation E_i, i.e., $P(E_i) = x_i$ for $E_i \in E$.
Output: Overall probability for filer expression, i.e., $P(E)$.

```
function getProbability(E)
    E_left, op, E_right ← E
    if op is boolean logical operator then
        return x_i
    else if op is binary operator then
        P(E_left) = getProbability(E_left)
        P(E_right) = getProbability(E_right)
        if op == AND then
            return P(E_left) × P(E_right)
        else if op == OR then
            return 1 − P(E_left) × P(E_right)
        end if
    end if
end function
```

3.2.2 Evaluate a filter expression. During data generation process, after the non-key columns are generated, we need to check if the corresponding filter expression is satisfied for each query in order to control the join distribution. Here we propose algorithm 2 to evaluate the complex filter expression and return a boolean value. Let $f(E)$ represents a boolean value after evaluating filter expression E. If E only contains one filter operation, we pass the generated data value to the evaluate the filter operation. The proof of this algorithm is similar to the proof for algorithm 1, where we start from a single filter operation and build induction on that.

Algorithm 2 Determine if a filter expression is satisfied with generated data

Input: Filter expression E on table T;
A row record r for table T with generated value for all attributes involved in E.
Output: A boolean value $f(E)$ represents whether E is satisfied or not. i.e., $f(E) = 1$ if E is satisfied, otherwise 0.

```
function isSatisfied(E)
    E_left, op, E_right ← E
    if op == AND then
        return f(E_left) · f(E_right)
    else if op == OR then
        return f(E_left) + f(E_right)
    end if
end function
```

3.3 Set Operations

Query set operations can be performed on more than one query, treating results from two queries separately as two sets. We use the query analyzer strategy mentioned above and collect cardinality constraints from each query. However, unlike the simple query case discussed in Section 3.1, extra logic is required to ensure that the resulting cardinality is non-zero after performing the set operation on the results of each query.

Notice that while set operators only require to operate on relations with the same number of attributes with corresponding same attribute datatype respectively, our algorithm cannot accommodate non-key columns originating from different relations. The algorithm relies on the assumption that queries involved in the set operation will either return the same attributes from same table(s) or primary/foreign-key attributes after projection.

There are three keywords for set operations in SQL language connect a pair of queries: `UNION`, `EXCEPT`, and `INTERSECT`. We introduce the analysis algorithm for each case in detail the following section, and summarize at the end in table 2.

3.3.1 Set Union. In the context of queries results union, it is straightforward that if we satisfy the cardinality constraint of both queries, then their union will contain results. Thus, we can treat them as distinct queries, analyze each one separately, and ensure that the selectivity value is greater than zero. i.e., the cardinality constraints for query $Q = Q_1 \cup Q_2$ is $C = C_1 \cup C_2$, where C_1 and C_2 are cardinality constraints for Q_1 and Q_2 separately.

3.3.2 Set Difference. The `EXCEPT` keyword connecting two queries Q_1 and Q_2 returns all rows from result of Q_1 but are not from Q_2 [3], which is also referred as set difference between two queries. Here, our goal is to ensure that the the set difference between first query Q_1 and second query $|Q_2|$ is not empty, i.e., $|Q| = |Q_1| - |Q_2| > 0$.

To achieve this, we transform both queries into a set of cardinality constraints. Let C_1, C_2 be the cardinality sets for Q_1 and Q_2 respectively. We will keep the selectivity for the first query while change the selectivity for the second one to be a value close to 0 (default to 0). i.e, Change selectivity s to $0\ \forall\ c_i \in C_2$. Then, we take the union of cardinality constraints as $C = C_1 \cup C_2$. For example, after analyzing the query in listing 3, we will get the following cardinality constraints:

- c_1 = (users, users.login `IN` (P1, P2), 0.5)
- c_2 = (users, users.age > P3, 0.0)
- c_3 = (users, users `JOIN` projects `ON` id = user_id)
- c_4 = (projects, users `JOIN` projects `ON` id = user_id, 0.0)

Among $c_i \in C$, $c_1 \in C_1$, and $c_2, c_3, c_4 \in C_2$. We can see that only c_1 has a selectivity bigger than 0.

Listing 3: Queries with except operations

```
(SELECT users.* FROM users WHERE users.login IN (P1, P2))
EXCEPT
(SELECT users.* FROM users INNER JOIN projects ON users.
     id = projects.user_is WHERE users.age > P3);
```

3.3.3 Set Intersection. The case of two queries connected by `INTERSECT` keyword presents a different challenge from the previous two cases,

Table 2: Set Operation Analyzing Methods

Set Operation	How to extract cardinality constraint
Union	$C = C_1 \cup C_2$
Except	$C = C_1 \cup C_2$ with $s = 0 \; \forall s \in C_2$
Intersect	$C = C'$ where C' is for transformed query Q'

as it requires that the intersection of the query results is non-zero. That is, given two queries, Q_1 and Q_2, our goal is to ensure $|Q_1 \cap Q_2| > 0$.

To achieve this, we analyze the cardinality constraints from both queries and combine them together. Notice that this is different from take a union set of C_1 and C_2, as the filter operations are compressed into one cardinality constraint for each table. Also, if different tables are introduced from Q_1 and Q_2, we will put a join cardinality constraint to express their overlapping results. The methods here only work under our assumption that the projection attributes from Q_1 and Q_2 are the same or they are corresponding primary/foreign key pairs. This method does not apply to the general "union-compatible" attributes, meaning that they should return the same number of columns with compatible data types [3]. For example, we can extract cardinality constraints from query in listing 4 as the following (set selectivity as default value $s = 0.5$):

- c_1 = (users, users.login IN (P1, P2) AND users.age > P3, 0.5)
- c_2 = (users, users JOIN projects ON id = user_id)
- c_3 = (projects, users JOIN projects ON id = user_id, 0.5)

Listing 4: Queries with intersect operations

```
(SELECT users.* FROM users WHERE users.login IN (P1, P2))
      INTERSECT
(SELECT users.* FROM users INNER JOIN projects ON users.
     id = projects.user_is WHERE users.age > P3);

-- transformed query Q'
SELECT users.* FROM users
INNER JOIN projects ON users.id = projects.user_id
WHERE users.login IN (P1, P2) AND users.age > P3;
```

Notice that $c_1 \notin C_1$ and $c_1 \notin C_2$, rather, c_1 combines the filter constraints from Q_1 and Q_2. In face, $C' = \{c_1, c_2, c_3\}$ represents the cardinality constraint for the transformed query Q'. Since Q' represents the intersection part from the original set operation query Q, $|Q'| > 0 \Rightarrow |Q| > 0$.

In summary, the overall strategy to deal with set operations is summarized in table 2.

4 SUBQUERY ANALYZER OVERVIEW

Subquery analyzer is a part of query analyzer that deals with subqueries. Same as the previous section, the goal of subquery analyzer is taking SQL queries and analyze them into cardinality constraints for data generator. Because subquery contains nested query structure, generating temporary views on the fly to perform operations in multiple steps for the main query, it is not as easy to extract filter and join cardinality constraints from subqueries. Therefore, for subqueries, we come up with a case-by-case method that first finds an alternative simple query Q', then analyze and extract cardinality constraints from Q' for the data generator. We will argue that as long as the cardinality constraints C' of alternative query Q' can be satisfied, the cardinality of original query results is non-zero, i.e., $|Q| > 0$.

Table 3: Subquery cases

Pattern	Example	Characteristics
Subquery Expression	(NOT) EXISTS, (NOT) IN, ANY, ALL	Return a boolean result
Table Expressions	FROM	Return a virtual table
Value Expressions	SELECT, aggregation	Return a scalar value

Let Q be a query that contains at lease 1 subquery. While different rules are applied for each subquery case, the general process is described as the following:

(1) Analyze input query Q, decide which subquery case(s) apply to Q;
(2) Based on the rules for each case, return a list of cardinality constraints;
(3) Check any extra logical to combine with the outer main query;
(4) Return a list of cardinality constraints.

The subquery type is divided into three fundamental categories: subquery for row comparison expression, subquery for value expression, and subquery for table expression. For each case, we summarize their keywords patterns and characteristics in table 3.

Let ANALYZE(Q) be the general function that analyzes a query. The overall algorithm of ANALYZE is illustrated in algorithm 3, in which we first detecting all possible cases of set operations or subqueries, then apply corresponding rules on each of them to return cardinality constraints. If there are nested queries or multi-layer nested subqueries, algorithm in function ANALYZE recursively applies the same logic to each. In the following sections, we provide detailed explanations for each subquery category and their corresponding analyze rules.

Algorithm 3 Overall algorithm to process a query

Input: A SQL query Q
Output: A list of cardinality constraints C
  function ANALYZE(Q)
    $C = \{\}$
    if $Q \in$ simple query **then**
      C+ = ANALYZE(Q)
    else if $Q \in$ set operation query **then**
      for each $q \in Q$ **do**
        C+ = ANALYZE(Q)
      end for
    else if $Q \in$ subquery **then**
      $S \leftarrow Q$ ▷ Get all subquery clauses from original query
      for each $s \in S$ **do**
        C+ = ANALYZE(s)
      end for
    end if
    return C
  end function

5 SUBQUERY FOR ROW COMPARISON EXPRESSION

In this section, we discuss the method to analyze subquery into cardinality constraints for row comparison expressions. Row comparison expressions are the query clause that follows `WHERE` and `HAVING` keywords. The expression will return a boolean (true/false) value to decide if the filter condition is satisfied for a query result row. A subquery might embed after row comparison keywords including `(NOT) IN`, `(NOT) EXISTS`, `ANY`, `SOME`, `ALL`.

5.1 Semi-join Transformation

EXISTS and IN are boolean operators that accept a subquery argument in the format of `op (subquery)`. Boolean operators evaluate to true depending on the results returned by the subquery. One example of subqueries following the boolean operator is shown in list 6. Let S denotes the subquery in Q, one key insight is that S imposes extra filter constraints and join constraints for main query Q to find a match on table attributes. By leveraging this insight, we can transform original query Q with nested query structure into an alternative query Q' that incorporate constraints derived from the subquery S. Let T_1 be the table set that is involved in outer main query, T_2 be the table set that is involved in inner subquery, C be the returned cardinality constraints, and ANALYZE be the function that analyzes a query for general case. The algorithm is shown in 4, which we first check if there is any additional tables in S, then extract cardinality constraints.

Algorithm 4 Extract cardinality constraints from a subquery following `EXISTS`

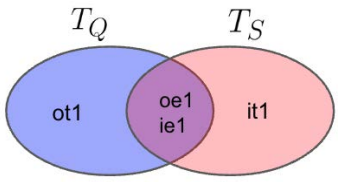
Input: A query Q contains subquery S with pattern `EXISTS (S)`
Output: A list of cardinality constraints C

1: **function** ANALYZEEXISTS(Q)
2: $T_Q \leftarrow$ a set of tables in outer main query
3: $T_S \leftarrow$ a set of tables in subquery S
4: **if** $T_Q \cap T_S = \emptyset$ **then**
5: C_S = ANALYZE(S) ▷ Treat S as a separate query
6: **return** $C = C_Q \cup C_S$
7: **else if** $T_Q = T_S$ **then**
8: Combine filter cardinality constraints from S to get C.
9: **return** C.
10: **else if** $T_Q \cap T_S \neq \emptyset$ **then**
11: Get alternative Q' from Q by semi-join transformation.
12: Extract C' from Q'.
13: **return** $C_{Q'}$.
14: **end if**
15: **end function**

Semi-join Transformation in algorithm 4 line 11 is a common pattern used in query prepare-time optimization for better execution performance. When $T_S \cap T_Q \neq \emptyset$, the outer query refers to one or more columns from tables that are only available in subquery to perform row comparison. For such case, we call the subquery S is not *independent* from the main query.

Definition 5.1. For a query Q that contains a subquery S, we call the subquery S is *independent* from the main query Q if S does not refer any column from Q and outer query does not refer to any columns that are only available in subquery S. I.e., S can be execute without the context of the main query Q.

Figure 3: Venn diagram for semi-join transformation



The general query pattern where the subquery S is dependent from the main query is shown in listing 5, where ot_i and it_i represents outer tables and inner tables respectively, oe_i and ie_i represents expressions refers to attributes in outer and inner table, and op $\in$ {`EXIST`, `IN`}. Let $T_Q = \{ot_i\}$ be the set of tables that are only used in the outer query, and $T_S = \{it_i\}$ be the set of tables that are only used in subquery S. S is *dependent* from Q implies that $T_Q \cap T_S \neq \emptyset$, and one or more attributes from T_S are compared or matched values with attributes from T_Q. The relationship of sets of T_Q and T_S is shown in figure 3, and this association can also be articulated via semi-join operation on T_Q.

Listing 5: General pattern for Query Satisfies a JOIN transformation

```
-- Q: original query
SELECT ... FROM ot1, ... WHERE (oe1, ...) op (SELECT ie1,
      ... FROM it1, ...);
-- Q': semi-join transformed query
SELECT ... FROM ot1, it1, ... WHERE (oe1, ie1, ...);
```

We provide an example from listing 6 for better illustration. Here, T_1 = { products }, T_2 = { products, suppliers }, so $T_1 \cap T_2$ = { products } $\neq \emptyset$. Apply the pattern from listing 6, we then can transform Q into Q' by build a `JOIN` relationship on the key attributes products.id and suppliers.product_id. Then, we analyze cardinality constraints C' from Q' as the following (using default selectivity $s = 0.5$:

- c_1 = (products, price < 20, 0.5)
- c_2 = (products, products `JOIN` suppliers `ON` products.id = suppliers.product_id)
- c_3 = (suppliers, products `JOIN` suppliers `ON` products.id = suppliers.product_id, 0.5)

For `IN` keyword, we will use the exactly same logic mentioned above in algorithm 4 except switch the operation keyword to `IN`.

Listing 6: Example of subquery lives EXISTS expression

```
-- original query Q with nested query structure
SELECT * FROM suppliers WHERE EXISTS
(SELECT products.name FROM products WHERE products.id =
     suppliers.product_id AND products.price < 20);

-- alternative query Q' without subquery inside
SELECT * FROM products JOIN suppliers ON products.id =
     suppliers.product_id AND products.price < 20;
```

5.2 Anti-join Transformation

Compared with EXISTS and IN, boolean compactors NOT EXISTS and NOT IN followed by a subquery describes an anti-join operation, which returns no match for the target attribute. Our objective is to ensure that the final outcome of these queries is non-zero through analyzing the query into cardinality constraints. As shown in listing 5, the overall logic for analyzing subquery after NOT EXISTS and NOT IN key words is very similar to the previous section, but with a twist of negating conditions in S to ensure the results from subquery to be empty in line 6, 9, and 12.

Algorithm 5 Extract cardinality constraints from a subquery following NOT EXISTS

Input: A query Q contains subquery S with pattern NOT EXISTS (S)
Output: A list of cardinality constraints C

1: **function** ANALYZENOTEXISTS(Q)
2: $T_Q \leftarrow$ a set of tables in outer main query
3: $T_S \leftarrow$ a set of tables in subquery S
4: **if** $T_Q \cap T_S = \emptyset$ **then**
5: C_S = ANALYZE(S) ▷ Treat S as a separate query
6: Set all selectivity s in C_S to 0.
7: **return** $C = C_Q \cup C_S$.
8: **else if** $T_Q = T_S$ **then**
9: Negate all operators for filter cardinality constraints from S, combine S to outer query constraints to get C.
10: **return** C.
11: **else if** $T_Q \cap T_S \neq \emptyset$ **then**
12: Get alternative Q' from Q by anti-join transformation.
13: Extract C' from Q'.
14: **return** C'.
15: **end if**
16: **end function**

Anti-join Transformation is similar to semi-join transformation. For a subquery S that is *dependent* from main outer query Q, we can "flatten" subquery S through JOIN operations to Q' with negating all filter conditions from S. For example, one can extract cardinality constraints from listing 7 as the following, which corresponds to an alternative query Q' without nested subquery structure.

- c_1 = (products, NOT (products.price < 20), 0.5)
- c_2 = (products, products JOIN suppliers ON products.id = suppliers.product_id)
- c_3 = (suppliers, products JOIN suppliers ON products.id = suppliers.product_id, 0.5)

Listing 7: Example of subquery lives EXISTS expression

```
-- original query Q with nested query structure
SELECT * FROM suppliers WHERE NOT EXISTS
(SELECT products.name FROM products WHERE products.id =
     suppliers.product_id AND products.price < 20);

-- alternative query Q' without subquery inside
SELECT * FROM products JOIN suppliers ON products.id =
     suppliers.product_id AND NOT (products.price < 20);
```

5.3 Comparison Operator

In the case of subqueries following comparison operators, only aggregation formats (such as AVE, MIN, MAX, and SUM) are allowed for the subquery Q, as the subquery must produce a value expression for matching main query rows. The subquery is typically in the form of expression op (subquery), where the comparison operator op includes >, =, <, and so on. Notice that if the aggregation is MIN or MAX, we can think of the filter expression as sorting the column and return the top value, which would naturally return a non-zero result for the query. For other aggregation operations, in order to make the the final returned result is non-zero, we have to first obtain the results from subquery in order to take the aggregation results into account as a condition for outer main query. This observation gives us an intuition about the column generation order. When generating data at the run-time, the data generator will follow the column partial order implied by the subquery after comparison operators.

6 SUBQUERY FOR VALUE EXPRESSION

In this section, we argue that subquery for value expression will not influence the final result size of a query, therefore can be safely disregarded. In a query statement, value expression is used for listing or calculating the query results, such as a field selection expression or a aggregation function call [3]. If the value expression is in the form of a subquery, it can only appear in the following two patterns (Let S represents the subquery):

- SELECT (S)
- SELECT aggregation_function(S)

In the context of queries, a value expression typically yields a single value and is therefore also known as a scalar expression. It is important to note that a value expression operates only on the returned values and does not modify them. As a result, the focus of analysis should be on the input values to the value expression to ensure their relevance, while the analysis of the value expression itself may be disregarded.

7 SUBQUERY FOR TABLE EXPRESSION

Table expression returns an intermediate table view. We elaborate the analysis methods for these two patterns in the following such that the analyzed constraint can ensure the query result is non-zero.

One common pattern of subquery serves as table expression is in the format of expression FROM (subquery), as shown in listing 8 where a subquery S is utilized as a temporary derived table or a view expression. The outer main query will use final result table expression executed from subquery. To analyze subqery following FROM keyword, we address from its correlation with the outer main query. If there is no other new table introduced in the outer main query, then we only need to analyze all constraints from the table involved in subquery. The algorithm to analyze subquery following FROM keyword in shown in listing 6. We first check if a subquery S is independent from the outer main query Q (line 4), and if there are any additional predicate conditions or aggregations on the tables T_S employed by the inner subquery (line 6, 9). One key insight for the table expression subquery is that we can first materialize intermediate virtual table generated by the subquery

(i.e., `table_alias` in listing 6), populate data for it, then combine data from intermediate table with schema table at the end.

Listing 8: Common pattern for subquery following FROM keyword

```
SELECT ... FROM (SELECT ie1, ie2 FROM it1 WHERE
    predicate_i) as table_alias, ot1 WHERE predicate_o;
```

Algorithm 6 Extract cardinality constraints from a subquery following `FORM`

Input: A query Q contains subquery S with pattern `FROM` (S)
Output: A list of cardinality constraints C

function ANALYZEFROM(Q)
  $T_Q \leftarrow$ a set of tables in outer main query
  $T_S \leftarrow$ a set of tables in subquery S
  if $T_Q = \emptyset$ & no predicate on T_Q in outer query **then**
    return ANALYZE(S)
  else if $T_Q \neq \emptyset$ and outer predicate only involves T_Q **then**
    $C =$ANALYZE(S) $\cup$ ANALYZE(Q)
    return C
  else if outer predicate involved T_Q and T_S **then**
    Materialize virtual table generated by S as T_{alias}
    Transform Q to Q' with T_{alias}
    C = ANALYZE(Q')
    Combine data from T_{alias} to T_Q, T_S
    return C
  end if
end function

8 INCORPORATE DATA INTEGRITY

Besides cardinality requirement from query, generating meaningful data from web application database also requires the data to conform integrity constraints introduced by web application frameworks. As mentioned in section 2, data integrity introduced by web application framework matters a lot for data fidelity, and maintaining framework data integrity constraint can greatly improve the quality of generated data. We will leverage the integrity data constraints and constraint extractor introduced in [13] and [15] to help us further improve generated data quality.

For every data type within a database, a corresponding data generation function is present, designed to produce a data index, which is subsequently mapped to data from a particular value domain. With more data integrity constraint added, the value domain shrinks with more strict rules apply to the data generation function, as shown in table 4. We elaborate each integrity constraint in the following.

Unique. A unique constraint dictates that every value can only appear once within a corresponding table attribute. When a unique constraint is identified for a particular table attribute, the data generator establishes a set to monitor used values, thereby preventing the generation of duplicate values for that column.

Not Null. The not null constraint is relatively straightforward, limiting the table attribute from containing null values. To guarantee compliance with the not null constraint, the null ratio within the data generation function is set to zero.

Figure 4: Foreign key constraints example from Redmine.

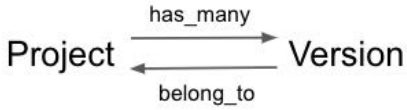


Table 4: Data Generation Function to conform Data Integrity Constraints

Data Integrity Constraint	Data Generation Function
Unique	Set a pool for used values
Not Null	Set Null ratio to be 0
Inclusion	Set value pool to be the fixed inclusion values
Foreign Key	Maintain generating partial order between attributes, set value pool from the primary-key column

Inclusion. In the case of an inclusion constraint, an attribute value is constrained to a limited set of values [15]. Upon the detection of an attribute with an inclusion constraint, the set of allowable values is provided and passed on to the data generation function. Within the data generation function, a value pool is established, guaranteeing that all generated data is selected from the inclusion value pool.

Foreign Key. For the attributes with foreign key constraints, their value depend on the primary key column in another table. In Ruby on Rails web framework, foreign key constraints is maintained by declaring `has_many` and `belong_to` key words. Figure 4 shows an example between the table Project and table Version. In this example, attribute "default_version" in table Project is refers to primary key column in table Version, and "project_id" attribute in table Version refers to primary key column in table Project. To incorporate foreign key constraints during run-time data generation, we only need to make sure columns with foreign key constraints are generated after the primary columns in another table, and then we can take values from those primary-key columns.

9 EVALUATION

For experiment setup, we run ezGen on a single MacBook Pro machine with Apple M1 chip and 8 GiB memory; the disk space is relatively limited (20GB left), which blocks running the very large scale experiments for data generation.

We select two workloads for experiment purpose:

- TPC-H, a decision-support benchmark and Redmine, a database-backed web application. TPC-H benchmark offers a set of bussiness oriented ad-hoc queries, and a database consists of 8 tables. The table size breakdown of TPC-H with sf=1 (scale factor) is shown in table 5, and TPC-H queries covers the majority scenario we covered in this paper, including queries that contain subquery structure following various keywords including `FROM`, `NOT IN`, `EXISTS`, etc.

Table 5: TPC-H Generation Table Size

Table Name	Size
customer	150,000
lineitem	6,000,000
nation	2,500
orders	1,500,000
part	200,000
partsupp	200,000
region	500
supplier	10,000

- Redmine is a popular project management web application with 3.7k GitHub stars, and it is built on the Ruby on Rails framework and comprises a total of 54 tables, with the average number of fields across all tables in Redmine as 8.

9.1 Comparison with Touchstone

We conducted a comparison of ezGen and Touchstone on the TPC-H benchmark with sf = 1. Figure 5 presents the results of our comparison. We acknowledge that Touchstone has limitations as it cannot take raw queries as inputs. Instead, users are required to manually convert queries into cardinality constraints. Our comparison results indicate that for Touchstone, queries 2, 4, 9, 11, and 15 have cardinality results of zero, primarily because these queries contain nested query structures that Touchstone fails to satisfy their cardinality requirements. On the contrast, ezGen is able to generate a database that not only satisfy cardinality requirements for simple queries but also for queries with embedded subqueries inside. Notice that y-axis in figure 5 is displayed in log-scaled due to the significant variation in query result sizes. This is because some queries are executes on very small tables while others on giant tables. For example, query 2 is executed on table *region*, which only contains 500 rows of records.

Furthermore, we compare end-to-end data generation time for Touchstone and ezGen. The total data generation time for Touchstone was 10.5s, while the total data generation time for ezGen was 10.7s, and hence the time for ezGen includes the overhead for parsing queries into cardinality constraints is very little.

9.2 Scalability Evaluation

Scalability on generated data size. In this experiment, we explore the scalability of ezGen as the size of requested data increases. We use TPC-H workload to generate data with sf=1, 10, 100, and 100. Larger scale factor is limited by the machine disk space. We compute the end-to-end data generation time, which includes time to analyze query into cardinality constraints, time to build column generator, and final data generation time. The results is shown in table 6, from which we can see the total time increases as data size increases in a linear rate, while other parts of ezGen (i.e., query analysis time and time to build column generator) remain relatively constant and constitute a minor proportion of the overall runtime.

Scalability on query workload size. To assess the scalability of ezGen in managing large-scale query workloads, we run experiments on Redmine. The queries are extracted from Redmine using a method proposed in [15], which involved executing test cases for different Redmine features and collecting the queries from the resulting log file. Approximately half of Redmine's query contains nested query structure as shown in 7. We set the data size for each table in Redmine to 2,000 and configured the selectivity of each query to be $s = 0.5$. The query analyze time is shown in figure 6. From the graph, we can see that the time query analyzer consumes is approximately linear with respect to the number of SQL queries.

Figure 5: Compare cardinality result: Touchstone v.s. ezGen

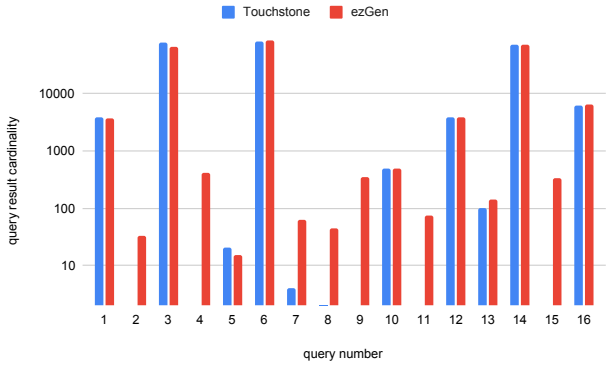
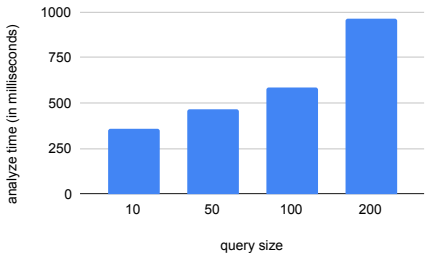


Table 6: TPC-H Data generation runtime breakdown (Unit: in seconds)

Scale Factor	Query Analyze Time	Building Generator Time	Generation Time	Total Time
1	0.11	0.03	10.39	10.54
10	0.10	0.03	89.86	90.00
20	0.23	0.05	181.98	182.26
100	0.11	0.08	891.98	892.19

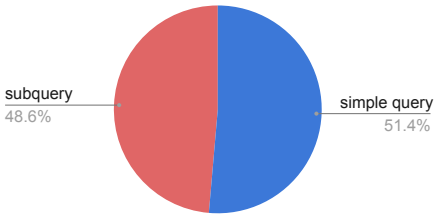
Figure 6: Query analyze time w.r.t. query size



10 CONCLUSION

This paper presents ezGen, a synthetic data generator that is capable of handling complex queries, including those featuring set operations and subqueries, as well as incorporating data integrity constraints from database-backed web applications.

Figure 7: Redmine query decomposition



We frame our data generating problem as an approximate probabilistic model, designed to fulfill cardinality requirements as defined by queries. Filter and join cardinality constraints are established, and the cardinality requirement of each query is converted into selectivity for each cardinality constraint. The definition of cardinality constraint and data generation method is closely connected as we need the filter cardinality constraints to build a data generator and tune the value distribution for each table attribute, then the join cardinality constraints ensures the join requirements are met during run-time data generation. Following the problem statement and definition is our detailed design for analysing complex queries into cardinality constraints. We introduced case-by-case rules for set operations and subqueries, with the key insights as using semi-join transformation and anti-join transformation. Then, we run experiements to show that ezGen can satisfy cardinality requirements for TPC-H database workload and web applications with a linear runtime with respect to data size and query size. To sum up, ezGen demonstrate great ability in terms of generating synthetic data for real-life application database.

REFERENCES

[1] [n.d.]. Django user manual. https://docs.djangoproject.com/en/. Last accessed 18 April 2023..

[2] [n.d.]. Hibernate user manual. https://hibernate.org/orm/documentation/. Last accessed 18 April 2023..

[3] [n.d.]. PostgreSQL manual. https://www.postgresql.org/docs/current/queries-union.html. Last accessed 18 April 2023..

[4] [n.d.]. Rails Backend Database. https://guides.rubyonrails.org/v2.3/getting_started.html#configuring-a-database. Last accessed 18 February 2023..

[5] [n.d.]. TPC-H. https://www.tpc.org/tpch/. Last accessed 18 April 2023..

[6] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data Generation Using Declarative Constraints. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1989323.1989395

[7] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

[8] Nedyalko Borisov and Shivnath Babu. 2011. Proactive Detection and Repair of Data Corruption: Towards a Hassle-Free Declarative Approach with Amulet. *Proc. VLDB Endow.* 4, 12 (aug 2011), 1403–1406. https://doi.org/10.14778/3402755.3402781

[9] Kenneth J. Dueker and J. Allison Butler. 1997. GIS-T ENTERPRISE DATA MODEL WITH SUGGESTED IMPLEMENTATION CHOICES.

[10] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity Estimation Using Probabilistic Models. *SIGMOD Rec.* 30, 2 (may 2001), 461–472. https://doi.org/10.1145/376284.375727

[11] Amir Gilad, Shweta Patwa, and Ashwin Machanavajjhala. 2021. Synthesizing Linked Data Under Cardinality and Integrity Constraints. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 619–631. https://doi.org/10.1145/3448016.3457242

[12] Joseph E. Hoag and Craig W. Thompson. 2007. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Rec.* 36, 1 (mar 2007), 19–24. https://doi.org/10.1145/1276301.1276305

[13] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. 2023. Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 632–645. https://doi.org/10.1145/3575693.3575699

[14] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 575–586. https://www.usenix.org/conference/atc18/presentation/li-yuming

[15] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2022. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. arXiv:2205.02954 [cs.DB]

[16] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating Databases for Query Workloads. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 848–859. https://doi.org/10.14778/1920841.1920950

[17] Luyi Qu, Yuming Li, Rong Zhang, Ting Chen, Ke Shu, Weining Qian, and Aoying Zhou. 2022. Application-Oriented Workload Generation for Transactional Database Performance Evaluation. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 420–432. https://doi.org/10.1109/ICDE53745.2022.00036

[18] D.M. Selfa, M. Carrillo, and M. Del Rocio Boone. 2006. A Database and Web Application Based on MVC Architecture. In *16th International Conference on Electronics, Communications and Computers (CONIELECOMP'06)*. 48–48. https://doi.org/10.1109/CONIELECOMP.2006.6

[19] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1542–1555. https://doi.org/10.1145/3514221.3526168