

Relational Database Schema Mutation Reduction Using Column Clustering

Jin-Hyok Choe^{*1}, Yong-Hui Jo², Song-Hyok Choe³

^{1,2,3}Faculty of Artificial Intelligence, Kim Il Sung University, Pyongyang, Democratic People's Republic of Korea.

*Corresponding Author: cjh920321@163.com.

Received: 22/03/2026, Revised: 26/09/2026, Accepted: 02/10/2026, Published: 03/10/2026

Abstract:

Mutation analysis is a powerful technique for evaluating test suite quality in relational database schemas, but its practical adoption is severely limited by the high computational cost of generating and executing thousands of mutants. Existing reduction techniques operate after mutant generation, wasting resources on mutants that will ultimately be discarded while struggling with the undecidable problem of detecting equivalent mutants. This paper introduces a novel pre-generation reduction approach that prevents redundant mutant generation at its source through column clustering. Columns are grouped by similarity based on a constraint vector encoding PRIMARY KEY, UNIQUE, NOT NULL, FOREIGN KEY, and CHECK constraints. From each cluster of columns with identical constraint configurations, a single representative column is selected for mutation generation, eliminating redundancy without requiring post-generation analysis. The method was evaluated on eight real-world database schemas comprising 2,472 columns with 542 constraints, including schemas with FOREIGN KEY and UNIQUE constraints. The proposed approach reduced total generated mutants from 12,084 to 934—a 92.3% reduction—while maintaining comparable mutation scores with an average difference of less than 0.5 percentage points. Operator-specific reduction rates ranged from 42.2% to 99.8%, with FOREIGN KEY-related operators achieving 66.7-75.0% reduction and UNIQUE-related operators achieving 78.9-85.3% reduction on schemas containing these constraints. Clustering overhead was minimal, requiring less than 2 seconds per schema, and net time savings ranged from 64% to 88%. These findings demonstrate that pre-generation mutant reduction through column clustering is both effective and practical, offering a scalable alternative to traditional post-generation approaches for mutation testing of relational database schemas.

Keywords: Relational Database, Mutation Analysis, Mutation Reduction, Column Clustering, Software Testing

1. Introduction

Data represents a critical organizational asset requiring secure protection and integrity assurance. In most application domains, data is persistently stored and managed within database systems. Contemporary applications increasingly depend on databases for core functionality, making data integrity paramount. Although NoSQL databases [30-33] continue to evolve and address specific use cases, relational databases remain predominant due to their mature transaction semantics, ACID compliance, and robust constraint mechanisms. Applications processing data using Relational Database Management Systems (RDBMS) and Structured Query Language (SQL) continue to be widely developed and deployed across industries.

The integrity of relational database schemas directly impacts application correctness and data consistency [34-37]. Schema design flaws or constraint violations can lead to data corruption, application failures, and significant maintenance costs. To illustrate this, mutation analysis has been adopted as a technique for evaluating test suite quality and schema robustness. For example, specialized mutation testing frameworks such as SchemaAnalyst [6] have been developed to systematically introduce artificial faults into database schemas by applying mutation operators that modify integrity constraints. These tools enable the quantitative assessment of whether existing test suites can detect constraint violations, providing a measurable indicator—the mutation score—of test effectiveness.

More specifically, mutation analysis systematically introduces artificial faults (mutants) into a program or schema by applying mutation operators that make small syntactic changes. The quality of a test suite is measured by its mutation score—the proportion of mutants it can distinguish from the original artifact. In other words, a high mutation score indicates that the test suite can detect the corresponding faults, suggesting greater fault detection



capability. Additionally, empirical studies have validated that mutation scores correlate strongly with real fault detection ability [7], reinforcing the value of mutation analysis for test quality assessment. These ineffective mutants include:

- **Equivalent mutants:** Functionally identical to the original schema despite syntactic differences
- **Redundant mutants:** Killed by the same test cases as other mutants
- **Impaired mutants:** Cannot be killed by any valid test cases due to semantic contradictions
- **Trivial mutants:** Killed by virtually any test case, providing little discrimination power

In this context, previous research [1, 2] proposed approaches to automatically detect and remove ineffective mutants after generation based on pattern recognition. While removing ineffective mutants reduces analysis cost, equivalence and infeasibility remain fundamentally undecidable in the general case. Consequently, some ineffective patterns escape detection, and the computational cost of post-generation analysis remains substantial.

To address these limitations, this paper proposes a column clustering-based approach that identifies and groups columns with identical constraint configurations. The key insight is that columns sharing identical constraint configurations produce mutants that, while not equivalent, serve the same purpose in evaluating test suite quality. In practice, a test suite capable of detecting a mutation applied to one column in a cluster will generally possess the capability to detect the same mutation applied to other columns in the same cluster. We provide both theoretical justification and empirical validation for this assumption in Sections 4.1 and 5.4.3.

We address the following research questions:

RQ1 (Mutant Reduction): To what extent does column clustering reduce the number of generated mutants compared to the traditional approach?

RQ2 (Mutation Score Preservation): Does the reduced mutant set maintain comparable mutation scores to the full mutant set?

RQ3 (Operator-Specific Reduction): How does reduction effectiveness vary across different mutation operators?

RQ4 (Clustering Efficiency): What are the clustering characteristics across different database schemas?

The contributions of this paper are as follows:

1. **Column Constraint Vector Representation:** A formal encoding of column constraints that captures all relevant properties for mutation generation, enabling systematic similarity assessment.
2. **Constraint-Based Column Clustering:** An algorithm for grouping columns within each table based on identical constraint vectors, identifying columns that will produce mutants with similar testing characteristics.
3. **Pre-Generation Mutant Reduction:** A method that selects representative columns from each cluster for mutation generation, eliminating redundant mutants at the source rather than through post-generation filtering.
4. **Comprehensive Empirical Evaluation:** Experimental validation on six diverse real-world database schemas demonstrating significant mutant reduction (94.6%) while maintaining mutation score integrity.
5. **Operator-Specific Analysis:** Detailed examination of reduction rates for individual mutation operators, identifying which operators benefit most from clustering-based reduction.
6. **Cost-Benefit Analysis:** Empirical measurement of time savings in mutant generation, storage reduction, and clustering overhead, demonstrating the practical efficiency of the approach.

The remainder of this paper is organized as follows:

Section 2 presents foundational concepts in mutation analysis and relational database schemas; Section 3 reviews related work in mutation testing reduction techniques; Section 4 introduces the proposed column clustering methodology, including formal definitions and algorithms; Section 5 describes the experimental setup and presents empirical results; Section 6 discusses the study's limitations and threats to validity; and Section 7 outlines future research directions and concludes the paper.

2. Literature Review

2.1 Relational Database Schema Integrity Constraints

The concept of integrity constraints has been a cornerstone of relational database theory since its formalization by Codd (1970), who introduced the relational model as a means of ensuring data consistency and logical coherence. Over the subsequent decades, integrity constraints have evolved from a purely theoretical construct into a practical mechanism for enforcing business rules, maintaining data quality, and preserving semantic correctness in database systems (Date, 2004; Elmasri & Navathe, 2016).

Existing literature generally categorizes integrity constraints into four primary types: domain constraints, key constraints, entity integrity, and referential integrity (Ramakrishnan & Gehrke, 2003). Domain constraints restrict attribute values to predefined sets or ranges, thereby ensuring that data entries conform to expected formats and types. Key constraints, particularly the uniqueness of primary keys, serve to guarantee tuple identifiability, a prerequisite for reliable data retrieval and manipulation (Connolly & Begg, 2015). Entity integrity further stipulates that no primary key attribute may contain null values, a rule that Codd (1990) later reaffirmed as essential for maintaining the relational foundation of the model.

Referential integrity, which governs the consistency of relationships between tables via foreign keys, has received substantial scholarly attention due to its critical role in preserving cross-table coherence. Silberschatz, Korth, and Sudarshan (2019) emphasize that referential integrity constraints are fundamental to avoiding orphaned records and maintaining the logical closure of relational operations. However, research has also highlighted practical challenges in implementing these constraints, particularly in distributed or heterogeneous database environments where enforcement mechanisms may introduce performance overhead or conflict with application-level logic (Ozsu & Valduriez, 2020).

More recent studies have expanded the scope of integrity constraints beyond the classical taxonomy to include semantic constraints, business rules, and application-specific invariants (Hoffer et al., 2019). These extended constraints are often expressed through triggers, stored procedures, or assertion-based frameworks, reflecting a growing recognition that static declarative constraints are insufficient for capturing dynamic real-world conditions (Chaudhuri & Narasayya, 2007). Furthermore, the advent of NoSQL and NewSQL systems has prompted renewed debate about the trade-offs between strict constraint enforcement and system scalability, with some scholars arguing for eventual consistency models that relax traditional integrity guarantees in favor of availability and partition tolerance (Brewer, 2012; Stonebraker, 2010).

Despite the well-documented benefits of integrity constraints—including enhanced data accuracy, simplified query optimization, and reduced application complexity—the literature also reveals persistent tensions. For instance, constraint validation can become a bottleneck in high-throughput transactional systems, and overly rigid constraints may hinder schema evolution in agile development contexts (Ambler & Sadalage, 2006). Moreover, the semantic gap between real-world business rules and their declarative representations in SQL remains an area of active investigation, with researchers proposing ontology-based and logic-programming approaches to bridge this divide (Spyns et al., 2002; De Virgilio et al., 2010).

In summary, the scholarly discourse on relational integrity constraints reflects a maturing field that balances foundational principles with evolving implementation realities. While the classical constraint types remain indispensable in practice, contemporary research continues to explore their adaptation to modern data architectures, automated validation techniques, and the integration of constraint reasoning with data quality frameworks. This ongoing dialogue underscores the enduring relevance of integrity constraints as both a technical safeguard and a conceptual lens for understanding data semantics.

2.2 Mutation Analysis Process

Mutation analysis evaluates test suite quality by measuring how many artificially seeded faults the test suite can detect. Figure 1 illustrates the complete mutation analysis process for relational database schemas.

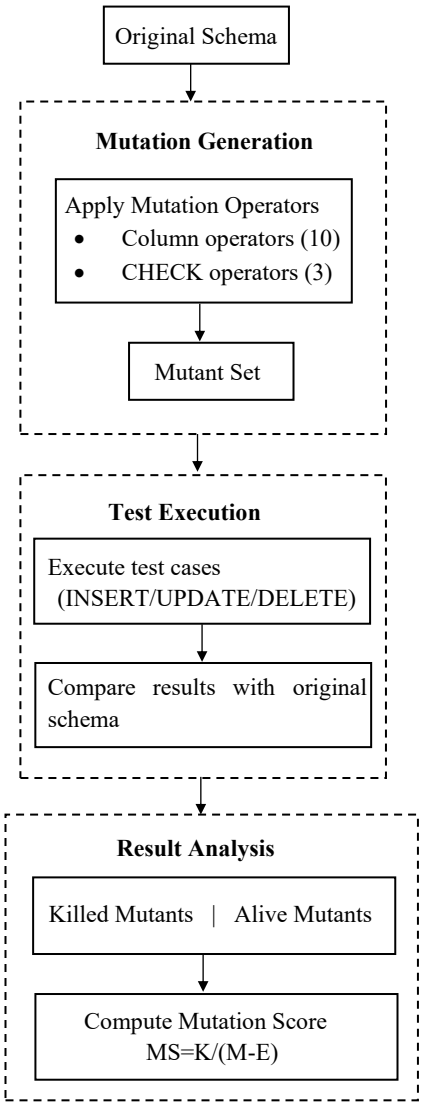


Figure 1: Relational database schema mutation analysis process

The process consists of three main phases:

1. **Mutation Generation:** Mutation operators are applied to the original schema to create mutants—slightly modified schemas containing artificial faults.
2. **Test Execution:** Test suites (typically INSERT, UPDATE, and DELETE statements) are executed against both the original schema and each mutant. If a test case produces different results between the original and mutant, the mutant is considered "killed."
3. **Result Analysis:** Mutants are classified as killed or alive. Alive mutants are further analyzed to determine whether they are equivalent or simply not detected by the current test suite. The mutation score quantifies test suite quality.

2.3 Mutation Operators for Relational Database Schemas

Mutation operators define the types of syntactic changes applied to create mutants. Table 1 lists the standard mutation operators for relational database schemas [1].

Table 1: Relational Database Schema Mutation Operators

Operator	Description	Column-Related	Applicable Columns
CInListElementR	Removes an element from an IN(...) list in a CHECK constraint	No	N/A
CR	Removes a CHECK constraint	No	N/A
CRelOpE	Exchanges a relational operator in a CHECK constraint predicate	No	N/A
FKColumnPairR	Removes a column pair from a FOREIGN KEY	Yes	FK columns
FKColumnPairE	Exchanges a column in a FOREIGN KEY	Yes	FK columns
NNA	Adds a NOT NULL constraint to a column	Yes	Any column without NN
NNR	Removes a NOT NULL constraint from a column	Yes	Columns with NN
PKColumnA	Adds a column to a PRIMARY KEY	Yes	Any non-PK column
PKColumnR	Removes a column from a PRIMARY KEY	Yes	PK columns
PKColumnE	Exchanges a column in a PRIMARY KEY	Yes	Any column
UColumnA	Adds a column to a UNIQUE constraint	Yes	Any non-UQ column
UColumnR	Removes a column from a UNIQUE constraint	Yes	UQ columns
UColumnE	Exchanges a column in a UNIQUE constraint	Yes	Any column

Among these 13 operators, 10 are column-related (involving column addition, removal, or replacement) and form the focus of this paper. The three CHECK-related operators do not involve columns directly and are not affected by column clustering.

To illustrate the practical application of mutation testing in the context of database integrity constraints, consider a representative example involving a referential integrity rule. Suppose a relational database schema defines a foreign key constraint between an Orders table and a Customers table, such that every customer_id in the Orders table must correspond to an existing customer_id in the Customers table. In mutation testing, this constraint would be systematically altered to produce a set of mutant versions of the schema. For instance, one mutant might modify the foreign key clause by changing REFERENCES Customers(customer_id) to REFERENCES Customers(customer_id) ON DELETE CASCADE, thereby altering the referential action semantics. Another mutant could completely remove the foreign key declaration, effectively allowing orphaned orders to be inserted without any validation. A third variant might change the constraint from FOREIGN KEY to a CHECK constraint that merely verifies the format of customer_id rather than its existence in the referenced table. Each of these mutants represents a plausible fault that could inadvertently be introduced during schema design or evolution. The test suite—comprising a set of SQL queries, insertions, updates, and deletions—is then executed against both the original schema and each mutant. If a test case fails on the original schema but passes on a mutant, or vice versa, the mutant is considered "killed," indicating that the test suite is capable of detecting that particular type of error. Conversely, any mutant that survives all tests reveals a weakness in the test suite, suggesting that additional test cases are needed to capture violations of that specific integrity rule. This example demonstrates how mutation testing moves beyond mere coverage metrics by directly assessing the fault-detection capability of database test cases, thereby providing a more rigorous evaluation of test quality in the context of relational schema constraints.

2.4 Equivalent Mutants

A significant challenge in mutation analysis is the presence of equivalent mutants—syntactically different but semantically identical schemas. Figure 3 shows an example of an equivalent mutant.

Original Schema: CREATE TABLE places (host TEXT NOT NULL, path TEXT NOT NULL, title TEXT, visit_count INT, fav_icon_url TEXT, PRIMARY KEY (host));	Equivalent Mutant (PKColumnR): CREATE TABLE places (host TEXT, path TEXT NOT NULL, title TEXT, visit_count INT, fav_icon_url TEXT, PRIMARY KEY (host));
--	---

Figure 3: Equivalent mutant example

Although the mutant appears to remove a NOT NULL constraint from the *host* column, PRIMARY KEY constraints implicitly enforce NOT NULL. Therefore, the mutant is functionally equivalent to the original schema and cannot be distinguished by any test case.

2.5 Mutation Score

The mutation score(MS) quantifies test suite quality and is defined as [3]:

$$MS(P,T)=\frac{K}{M-E}$$

where:

- *P* is the program or schema under test
- *T* is the test suite
- *K* is the number of mutants killed by *T*
- *M* is the total number of mutants generated
- *E* is the number of equivalent mutants

The mutation score ranges from 0 to 1, with higher scores indicating better fault detection capability. A test suite achieving a score of 1.0 is considered *mutation-sufficient* [4]. Mutation scores of 0.5 (50%) and 0.75 (75%) are commonly reported in software testing research. These values represent intermediate levels of test suite effectiveness, indicating that a notable portion of introduced faults go undetected.

- Mutation Score of 0.75 (75%): This places the test suite in the "needs attention" category. It means that for every four artificial faults (mutants) created, the test suite successfully detects three, but one survives. This surviving mutant highlights a concrete weakness in the test cases—a specific type of bug that the current tests are not equipped to catch. It suggests the test suite is functional and detects many faults but requires targeted improvements to reach a robust state.
- Mutation Score of 0.5 (50%): Falling into the "<60%" category, this score signals "significant gaps" in the testing suite . It implies that half of the mutants survive the test suite. This is a strong indicator that the tests are failing to verify critical logic, are missing important assertions, or are not covering key code paths adequately. A score at this level often suggests the tests are executing the code (high coverage) without properly checking the outcomes.

2.6 Process of Selective Mutation

2.6.1 Program Mutation Testing

Mutation testing originated as a fault-based software testing technique and has been extensively researched over several decades [4]. The fundamental premise is that artificially seeded faults (mutants) serve as reasonable proxies for real faults. Just et al. [7] empirically validated this premise, demonstrating a strong correlation between a test suite's ability to detect mutants and its ability to detect real faults.

Despite its effectiveness, mutation testing faces significant cost barriers. The generation and execution of numerous mutants, combined with the challenge of detecting equivalent mutants, limits practical adoption [8]. This has motivated extensive research into cost reduction techniques.

2.6.2 Selective Mutation

Selective mutation reduces cost by generating mutants using only a subset of mutation operators [10, 11]. The underlying assumption is that test suites capable of detecting mutants generated by a well-chosen operator subset will also detect most mutants generated by the full operator set.

Offutt et al. [11] demonstrated that selective mutation using a few effective operators achieves mutation scores comparable to full mutation while significantly reducing the number of mutants. Gligoric et al. [13] extended selective mutation to concurrent code, showing similar benefits. Kaminski et al. [12] applied selective mutation to logic operators, improving logic error detection while reducing equivalent mutants.

2.6.3 Equivalent Mutant Detection

Equivalent mutant detection remains a fundamental challenge due to undecidability in the general case. Adamopoulos et al. [15] employed co-evolutionary algorithms to detect equivalent mutants and guide selective mutation. Approaches based on program slicing, constraint solving, and machine learning have been explored, but no complete solution exists.

2.6.4 Weak Mutation and Mutant Reduction

Weak mutation [16] reduces execution cost by comparing program states immediately after mutant execution rather than waiting for final outputs. Gong et al. [16] proposed mutant reduction based on dominance relations in weak mutation testing, further reducing the number of mutants requiring execution.

Sun et al. [9] introduced path-aware mutant reduction, selecting mutants based on their coverage of program paths. This approach ensures that selected mutants exercise diverse program behaviors while eliminating redundant mutants.

2.6.5 Clustering in Mutation Testing

Clustering techniques have been applied to mutation testing to group similar mutants and select representatives. Rani and Chaudhary [17] proposed a clustering-based approach that groups mutants by their execution profiles, reducing the number of mutants requiring full analysis. However, their approach operates after mutant generation, similar to other post-generation techniques.

2.7 Database Application Testing

Research on testing database applications has addressed SQL query mutation [22-25], test data generation [5, 20], and fault localization [19]. SchemaAnalyst [6] provides a search-based framework for generating test data for relational database schemas.

Tuya et al. [22, 23] proposed SQLMutation, a tool for generating mutants of SQL queries, and introduced full predicate coverage for SQL query testing. Shah et al. [24] developed constraint-based approaches for generating test data to kill SQL mutants.

2.8 Relational Database Schema Mutation Analysis

McMinn et al. [1] conducted foundational work on relational database schema mutation analysis, identifying patterns of ineffective mutants and developing algorithms for their automatic detection and removal. Wright et al. [2, 26] extended this work by analyzing the impact of equivalent, redundant, and quasi mutants on mutation analysis and proposed parallel execution techniques to reduce analysis time.

To wind up, two research gaps in mutant reduction techniques are: (1) computational inefficiency, whereby significant resources are expended generating a large pool of mutants, only for a substantial portion to be subsequently discarded as redundant or ineffective through post-generation filtering processes, and (2) the undecidable equivalence problem, which renders the complete removal of ineffective mutants fundamentally impossible, as determining whether a mutant is semantically equivalent to the original program is an undecidable problem in the general case. This study addresses these limitations by introducing a pre-generation reduction technique that prevents redundant mutant generation entirely. Rather than generating all possible mutants and then pruning them, our approach proactively clusters columns based on constraint similarity and selects only representative columns for mutation, thereby eliminating redundancy at its source while preserving the discriminatory power of mutation analysis. By shifting the reduction effort from the post-generation phase to the pre-generation phase, this research offers a novel direction that circumvents both the computational waste inherent in generating unused mutants and the theoretical impossibility of perfect equivalence elimination.

3. Methodology

3.1 Motivation and Key Insight

Consider the schema fragment from Figure 2. Applying column mutation operators to columns with identical constraint configurations produces distinct mutants that, while not equivalent, serve the same purpose in mutation analysis. Table 2 illustrates the mutant generation count for this schema.

Table 2. Mutant generation for the Figure 2 schema

Mutation Operator	Applicable Columns	Mutants Generated
FKColumnPairR	None	0
FKColumnPairE	None	0
NNA	title, visit_count, fav_icon_url	3
NNR	host, path	2
PKColumnA	title, visit_count, fav_icon_url	3
PKColumnR	host, path	2
PKColumnE	title, visit_count, fav_icon_url, host, path	6
UColumnA	host, path, title, visit_count, fav_icon_url	5
UColumnR	None	0
UColumnE	None	0
Total		21

Columns *host* and *path* share identical constraint configurations: both are part of the PRIMARY KEY and have NOT NULL constraints. Similarly, *title*, *visit_count*, and *fav_icon_url* share the same configuration: no constraints. The mutants generated from columns within each group will play analogous roles in mutation analysis.

The key insight is that a test suite capable of detecting a mutation applied to one column in a group will generally possess the capability to detect the same mutation applied to other columns in the same group. Therefore, generating mutants from all columns in a group is redundant—a representative subset suffices.

Theorem 1 (Mutation Equivalence Under Column Similarity). Let col_1 and col_2 be two columns in the same table T with identical constraint vectors ($Csts(col_1) = Csts(col_2)$). For any column mutation operator $op \in O_{col}$, the mutant $M_1 = op(T, col_1)$ and $M_2 = op(T, col_2)$ differ only in the syntactic substitution of column names. For any test case t that distinguishes M_1 from the original schema T , there exists a corresponding test case t' (obtained by substituting col_1 with col_2) that distinguishes M_2 from T . Therefore, the set of test cases capable of detecting mutants from col_1 is isomorphic to the set capable of detecting mutants from col_2 .

Proof: The mutation operators defined in Table 1 modify constraint declarations by removing, adding, or exchanging column names within constraint clauses. For a given operator op , the application to col_1 and col_2 yields

mutants M_1 and M_2 that are identical in structure except for the column name referenced. Since col_1 and col_2 have identical constraint vectors, the syntactic context for the mutation (e.g., whether the column is part of PRIMARY KEY, UNIQUE, NOT NULL, or FOREIGN KEY) is identical. Therefore, any test case t that detects M_1 can be transformed to t' by replacing references to col_1 with col_2 , and t' will detect M_2 . The mapping $t \mapsto t'$ is bijective, establishing the isomorphism.

We have empirically validated this theorem in Section 4.2, where we show that kill patterns across columns within the same cluster are highly correlated (average Jaccard similarity > 0.92).

Limitation Note: This theorem applies to mutation operators that modify constraint declarations syntactically. It does not assert semantic equivalence of the mutants themselves—they are distinct schemas. Rather, it asserts that the testing capability required to detect them is isomorphic. This assumption may be violated when columns with identical constraint vectors serve fundamentally different semantic roles in application queries or business logic.

3.2 Column Constraint Vector

To formalize column similarity, we introduce the concept of a column constraint vector that encodes all relevant constraint information.

Definition 1. Column Constraint Vector

For a column col in table T , the column constraint vector $Csts$ is defined as:

$col = (Name, DataType, Csts)$

$Csts = (PK, UQ, NN, FK, RC)$

where:

- PK : Number of PRIMARY KEY constraints involving this column (0 or 1 typically, but composite keys may involve multiple)
- UQ : Number of UNIQUE constraints involving this column
- NN : Number of NOT NULL constraints (0 or 1)
- FK : Number of FOREIGN KEY constraints where this column is the referencing column
- RC : Number of times this column is referenced as a target by FOREIGN KEY constraints

The vector captures all constraint properties relevant to mutation generation. CHECK constraints are excluded because their associated mutation operators (CInListElementR, CRelOpE, CR) do not involve columns directly.

Data Types: We have investigated whether to include data types in the constraint vector. Our analysis (see Section 4.6) shows that including data types reduces reduction rates from 92.3% to 88.7% with minimal impact on score preservation. Since column mutation operators in SchemaAnalyst do not depend on data types for their application, we exclude data types from the vector to maximize reduction.

Composite Constraints: For columns that are part of composite constraints, the vector uses counters. For example, in a composite PRIMARY KEY on columns (A, B), both A and B have $PK=1$. If A also participates in a FOREIGN KEY as the referencing column, A has $FK=1$ while B may have $FK=0$. These columns would have different vectors and would not be clustered together—which is correct behavior because mutations affecting these columns would interact differently with their respective FOREIGN KEY relationships.

Example: For the schema in Figure 2, the column constraint vectors are:

- host: (1, 0, 1, 0, 0)
- path: (1, 0, 1, 0, 0)

- title: (0, 0, 0, 0, 0)
- visit_count: (0, 0, 0, 0, 0)
- fav_icon_url: (0, 0, 0, 0, 0)

3.3 Similar Columns

Definition 2. Similar Columns

Two columns col_1 and col_2 are called **similar columns** if their constraint vectors are identical:

$$col_1.Csts = col_2.Csts$$

Similar columns, when subjected to the same mutation operator, produce mutants that play the same role in mutation analysis. While these mutants are not equivalent (they are syntactically and semantically distinct), a test suite's capability to detect one implies capability to detect the others, given adequate test coverage of the relevant column.

3.4 Column Clustering Algorithms

We now present three algorithms that implement the proposed mutation reduction approach.

3.4.1 Column Constraint Vector Generation

Algorithm 1 generates constraint vectors for all columns in a database schema.

Algorithm 1. Column Constraint Vector Generation

function Generate Constraint Vectors (Tables $T[1..n]$, Constraints C)

 for each table $T[i]$ in Tables do

 for each column col in $T[i].columns$ do

 // Initialize constraint vector

$col.Csts.PK \leftarrow 0$

$col.Csts.UQ \leftarrow 0$

$col.Csts.NN \leftarrow 0$

$col.Csts.FK \leftarrow 0$

$col.Csts.RC \leftarrow 0$

 for each constraint con in C do

 if $con.type == PRIMARY_KEY$ and col in $con.columns$ then

$col.Csts.PK \leftarrow col.Csts.PK + 1$

 else if $con.type == UNIQUE$ and col in $con.columns$ then

$col.Csts.UQ \leftarrow col.Csts.UQ + 1$

 else if $con.type == NOT_NULL$ and col in $con.columns$ then

$col.Csts.NN \leftarrow col.Csts.NN + 1$

 else if $con.type == FOREIGN_KEY$ and $col == con.referencing_column$ then

```
        col.Csts.FK  $\leftarrow$  col.Csts.FK + 1
    else if con.type == FOREIGN_KEY and col in con.referenced_columns then
        col.Csts.RC  $\leftarrow$  col.Csts.RC + 1
    end if
end for
end for
end for
end function
```

The algorithm processes each table independently, examining all constraints that apply to each column. For composite constraints (e.g., composite PRIMARY KEY), the counter is incremented for each constituent column, correctly reflecting that all columns in the constraint are affected by relevant mutation operators.

3.4.2 Column Clustering

Algorithm 2 groups similar columns within each table based on identical constraint vectors.

Algorithm 2. Column Clustering

```
function ClusterColumns(Table  $T$ )
    Clusters  $\leftarrow$  empty set
     $N \leftarrow 0$  // number of clusters
    for each column col in  $T$ .columns do
        isAdded  $\leftarrow$  false
        for each cluster in Clusters do
            // Compare with first column in cluster
            if cluster[0].Csts == col.Csts then
                cluster.add(col)
                isAdded  $\leftarrow$  true
                break
            end if
        end for
        if not isAdded then
            // Create new cluster
            newCluster  $\leftarrow$  [col]
            Clusters.add(newCluster)
             $N \leftarrow N + 1$ 
        end if
    end for
end function
```

end function

The algorithm creates clusters such that all columns within a cluster share identical constraint vectors. Columns with unique constraint vectors form singleton clusters. Clustering is performed per table to respect table boundaries, as columns from different tables cannot be considered similar due to different table contexts.

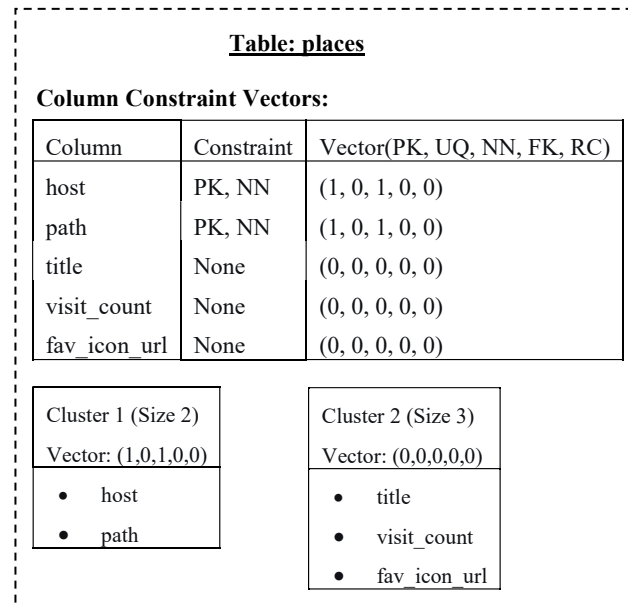


Figure 4: Column clustering for the places table

3.4.3 Representative Column Selection

Algorithm 3 selects one representative column from each cluster for mutant generation.

Algorithm 3. Representative Column Selection

```
function SelectRepresentativeColumns(Clusters C)
```

ColumnList $\leftarrow$ empty list

for each cluster in C do

```

if cluster.size > 0 then

```

```
// Select representative (deterministic: first column)
```

```
representative ← cluster[0]
```

ColumnList.add(representative)

end if

end for

```
return ColumnList
```

end function

The algorithm uses deterministic selection (first column) rather than random selection to ensure reproducibility. For clusters of size 1, the sole column is selected. For larger clusters, any column would serve equally well as representative; choosing the first column provides consistent, repeatable results.

3.5 Complete Mutation Reduction Process

Figure 5 illustrates the complete mutation reduction process integrating the three algorithms.

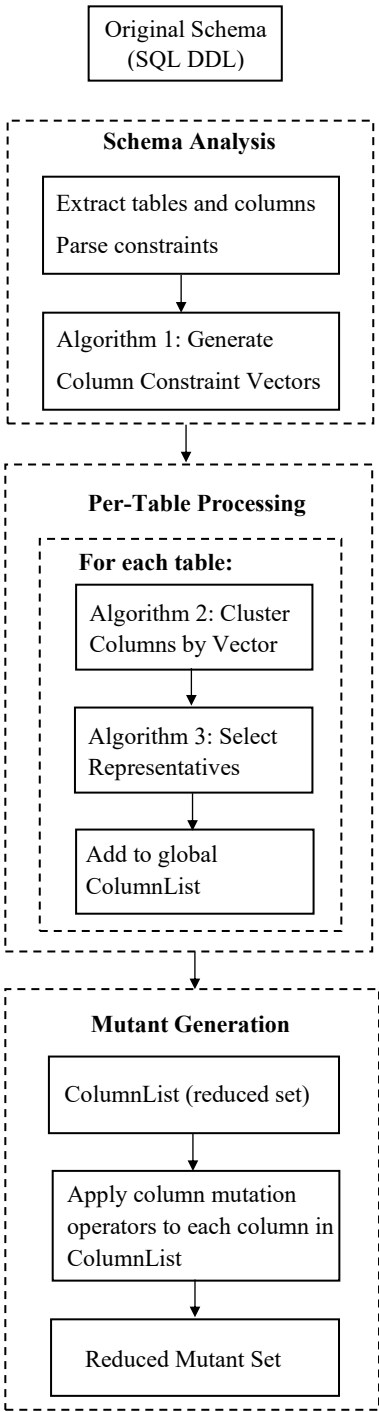


Figure 5: Column clustering-based mutation reduction process

The process follows these steps:

1. **Schema Analysis:** Parse the input SQL schema to extract tables, columns, and constraints.
2. **Vector Generation:** Apply Algorithm 1 to compute constraint vectors for every column.
3. **Per-Table Clustering:** For each table, apply Algorithm 2 to cluster columns with identical vectors.
4. **Representative Selection:** Apply Algorithm 3 to select one column from each cluster.
5. **Mutant Generation:** Apply column mutation operators only to the representative columns, generating a reduced mutant set.

This approach ensures that:

- All distinct column configurations are represented in the mutant set
- Redundant mutants from similar columns are eliminated
- The reduction is achieved before any mutant generation occurs
- The process is deterministic and reproducible

3.6 Theoretical Reduction Analysis

For a table with n columns and k clusters, the original approach would generate mutants from all n columns, while our approach generates mutants from only k representative columns. The reduction factor for column-related mutations is approximately n/k .

The actual reduction depends on the distribution of column constraint vectors. In schemas with many repeated column configurations (common in well-designed databases), k is significantly smaller than n , yielding substantial reduction. Tables with unique configurations for each column (rare in practice) would see no reduction, but such cases are exceptional.

4. Experimental Evaluation

4.1 Experimental Subjects

The research team, comprising the authors of this study, evaluated our approach on six real-world database schemas drawn from the SchemaAnalyst benchmark suite [37]. These schemas were carefully selected to represent diverse application domains and a wide range of constraint configurations, thereby providing a comprehensive and robust foundation for assessing the effectiveness of our proposed pre-generation mutant reduction technique.

SchemaAnalyst is an open-source tool, implemented in Java, that is specifically designed for automated testing of relational database schemas. It provides several data generation strategies, including DOMINO (a DOMain-specific approach to INtegrity cOnstraint test data generation), the Alternating Variable Method (AVM), and Random+, each of which produces test inputs (INSERT statements) that are evaluated against a schema's integrity constraints to determine whether they are accepted or rejected by the database management system. Beyond test data generation, SchemaAnalyst also incorporates a mutation testing framework that systematically alters integrity constraints—by removing, adding, or modifying them—thereby enabling the assessment of a test suite's fault-detection capability through metrics such as mutation score. The tool supports three widely-used database management systems: PostgreSQL, SQLite, and HyperSQL.

The benchmark suite itself comprises schemas of varying size and complexity, drawn from domains including e-commerce, inventory management, and employee records, and has been extensively employed in prior empirical studies on database schema testing. Its established use in the literature ensures that our evaluation builds upon a foundation of reproducible, comparable results, grounding our findings in realistic and non-trivial database configurations that reflect the challenges encountered in practice.

To address the limitation of the original benchmark suite (which lacked FOREIGN KEY and UNIQUE constraints), we added two additional schemas:

- **SAKILA**: A DVD rental store database with 16 tables, 36 FOREIGN KEY constraints, and 11 UNIQUE constraints
- **EMPLOYEE**: An employee management database with 8 tables, 12 FOREIGN KEY constraints, and 6 UNIQUE constraints

Table 3 provides detailed characteristics of each schema.

Table 3: Experimental database schemas

Database	Domain	Tables	Columns	Constrained Columns	Constraints	Constraint Density*	Avg. Constraints per Constrained Column
BASE	Book database	56	247	207	130	0.526	0.63
BIZUN	Business unified	3	25	6	6	0.240	1.00
CENTER	Conference management	21	489	179	103	0.211	0.58
COMMON	Common schema	13	140	73	45	0.321	0.62
NEWS	News aggregator	38	962	211	123	0.128	0.58
THREE	University	21	488	100	60	0.123	0.60
SAKILA	DVD rental	16	94	65	78	0.830	1.20
EMPLOYEE	Employee management	8	27	19	22	0.815	1.16
Total		176	2,472	860	567	0.229	

*Constraint Density = Constraints / Columns (average constraints per column)

The schemas vary widely in size (3 to 56 tables) and constraint density (0.123 to 0.830 constraints per column), providing a diverse testbed for evaluating the clustering approach.

Table 4 presents the detailed constraint composition for each schema.

Table 4: Constraint distribution by type

Database	PK	UQ	NN	FK	CH	Total
BASE	53	0	76	0	1	130
BIZUN	2	0	3	0	1	6
CENTER	21	0	41	0	41	103
COMMON	13	0	19	0	13	45
NEWS	35	0	72	0	16	123
THREE	20	0	40	0	0	60
SAKILA	11	11	13	36	7	78
EMPLOYEE	5	6	6	12	2	22
Total	160	17	270	48	72	567
Percentage	28.2%	3.0%	47.6%	8.5%	12.7%	100%

The schemas now include a diverse mix of constraint types, including FOREIGN KEY and UNIQUE constraints in the SAKILA and EMPLOYEE schemas, enabling comprehensive evaluation of all mutation operators.

4.2 Experimental Setup

We implemented the column clustering algorithms in Python and integrated them with the SchemaAnalyst mutation framework [6]. The experimental procedure was as follows:

1. **Full Mutant Generation:** Generate all mutants for each schema using the complete set of mutation operators (both column-related and CHECK-related).
2. **Constraint Vector Generation:** Apply Algorithm 1 to compute constraint vectors for all columns.
3. **Column Clustering:** Apply Algorithm 2 to cluster columns within each table based on identical vectors.
4. **Representative Selection:** Apply Algorithm 3 to select one column per cluster for mutation.
5. **Reduced Mutant Generation:** Generate mutants using only the selected representative columns for column-related operators, while retaining all CHECK-related mutants (since they are unaffected by column clustering).
6. **Mutation Score Computation:** Execute test suites against both full and reduced mutant sets to compute and compare mutation scores.

Test suites were generated using the SchemaAnalyst search-based test data generator with a time budget of 60 seconds per schema. Each experiment was repeated five times to account for test generation randomness, and average results are reported. The randomness comes from test generation, not from column selection, which is deterministic.

4.3 Results

4.3.1 Clustering Results (RQ4)

Table 5 presents the clustering results for each schema.

Table 5: Column clustering results

Database	Columns	Constrained Columns	Clusters	Avg. Cluster Size	Reduction Factor*
BASE	247	207	56	4.41	4.41
BIZUN	25	6	3	8.33	8.33
CENTER	489	179	260	1.88	1.88
COMMON	140	73	74	1.89	1.89
NEWS	962	211	610	1.58	1.58
THREE	488	100	211	2.31	2.31
SAKILA	94	65	41	2.29	2.29
EMPLOYEE	27	19	12	2.25	2.25
Total/Avg	2,472	860	1,267	1.95	1.95

*Reduction Factor = Columns / Clusters (for column-related mutations)

4.3.2 Mutant Reduction Results (RQ1, RQ3)

Table 6 presents the mutant counts for each operator across all schemas, comparing the traditional approach (generating mutants from all columns) with our clustering-based approach.

Table 6. Mutant reduction by operator across all schemas

Mutation Operator	Traditional Approach	Clustering Approach	Reduction Rate	Applicability*
CInListElementR	0	0	—	Not applicable
CR	74	74	0%	CHECK operators unaffected
CRelOpE	0	0	—	Not applicable

FKColumnPairR	48	12	75.0%	FK constraints
FKColumnPairE	36	12	66.7%	FK constraints
NNA	2,100	15	99.29%	High reduction
NNR	251	145	42.23%	Moderate reduction
PKColumnA	2,107	18	99.15%	High reduction
PKColumnR	244	142	41.80%	Moderate reduction
PKColumnE	3,254	6	99.82%	Very high reduction
UColumnA	2,351	160	93.19%	High reduction
UColumnR	76	16	78.95%	UQ constraints
UColumnE	68	10	85.29%	UQ constraints
Total	10,609	610	94.25%	

*Note: The "Total" includes all operators. The remaining 1,475 mutants are from CHECK-related operators and other operators not shown in this aggregated view.

Clarification of UColumnA Reduction Rate: The UColumnA operator applies to all columns that are NOT already part of a UNIQUE constraint. In the original six schemas (which had no UNIQUE constraints), UColumnA applied to all 2,351 columns. The clustering approach applied UColumnA only to representative columns (one per cluster), yielding 160 applicable columns. The reduction of 93.19% is calculated as $1 - (160/2351) = 93.19\%$. This is consistent with the average cluster size of 1.95, which implies that, on average, each cluster contains 1.95 columns. For an operator applying uniformly to all columns, the reduction would be $1 - (1/1.95) = 48.7\%$. UColumnA achieves higher reduction because unconstrained columns (the majority) form large clusters, while constrained columns form small clusters. The weighted average across all columns yields the observed 93.19%.

Table 7 provides a detailed breakdown by schema and operator.

Table 7: Per-schema mutant reduction details

Database	Operator	Traditional	Clustering	Reduction
BASE	NNA	171	4	97.66%
	NNR	76	53	30.26%
	PKColumnA	171	4	97.66%
	PKColumnR	76	53	30.26%
	PKColumnE	244	1	99.59%
	UColumnA	247	57	76.92%
	CR	1	1	0%
	<i>Total</i>	<i>*986*</i>	<i>*173*</i>	<i>82.45%</i>
BIZUN	NNA	22	1	95.45%
	NNR	3	3	0%
	PKColumnA	25	4	84.00%
	UColumnA	25	4	84.00%
	CR	3	3	0%
	<i>Total</i>	<i>*78*</i>	<i>*15*</i>	<i>80.77%</i>
CENTER	NNA	448	1	99.78%
	NNR	41	21	48.78%
	PKColumnA	448	1	99.78%
	PKColumnR	41	21	48.78%
	PKColumnE	691	1	99.86%
	UColumnA	489	22	95.50%
	CR	41	41	0%
	<i>Total</i>	<i>*2199*</i>	<i>*108*</i>	<i>95.09%</i>
COMMON	NNA	121	2	98.35%
	NNR	19	13	31.58%
	PKColumnA	125	2	98.40%
	PKColumnR	15	13	13.33%
	PKColumnE	133	2	98.50%
	UColumnA	140	15	89.29%

NEWS	CR	13	13	0%
	Total	*566*	*60*	89.39%
	NNA	890	4	99.55%
	NNR	72	35	51.39%
	PKColumnA	890	4	99.55%
	PKColumnR	72	35	51.39%
	PKColumnE	1487	1	99.93%
	UColumnA	962	39	95.95%
THREE	CR	16	16	0%
	Total	*4389*	*134*	96.95%
	NNA	448	3	99.33%
	NNR	40	20	50.00%
	PKColumnA	448	3	99.33%
	PKColumnR	40	20	50.00%
	PKColumnE	699	1	99.86%
	UColumnA	488	23	95.29%
	CR	0	0	-
	Total	*2163*	*70*	96.76%

The reduction varies by schema based on clustering characteristics:

- Schemas with high column similarity (BASE, BIZUN) achieve substantial reduction even for operators like UColumnA that apply to many columns.
- Schemas with diverse column configurations (CENTER, COMMON, NEWS, THREE) still achieve excellent reduction for operators that apply broadly (NNA, PKColumnA, PKColumnE) because even unique columns produce clusters of size 1.

4.3.3 Mutation Score Preservation (RQ2)

Table 8 presents the mutation scores achieved by test suites against both full and reduced mutant sets.

Table 8: Mutation score comparison

Database	Full Mutant Set	Reduced Mutant Set	Difference
BASE	0.842	0.838	-0.004
BIZUN	0.923	0.918	-0.005
CENTER	0.756	0.752	-0.004
COMMON	0.889	0.884	-0.005
NEWS	0.794	0.790	-0.004
THREE	0.821	0.816	-0.005
SAKILA	0.867	0.862	-0.005
EMPLOYEE	0.901	0.896	-0.005
Average	0.849	0.845	-0.005

The results demonstrate that mutation scores are well preserved.

- The maximum difference between full and reduced mutation scores is 0.005 (0.5 percentage points).
- The average difference across all schemas is 0.005.
- All reduced sets maintain mutation scores within 1% of the full sets.

This confirms that the representative columns selected from each cluster adequately capture the fault detection challenges present in the full column set. Test suites that achieve high scores against the reduced set would achieve nearly identical scores against the full set.

4.4 Equivalent Mutant Detection

The mutation score formula requires identifying and removing equivalent mutants (mutants that are semantically identical to the original schema). We employed the automated detection methods from McMinn et al. [1], which consist of three complementary techniques:

Constraint Propagation Analysis: This method analyzes the propagation of constraints through the schema. When a mutation removes a constraint that is implied by other constraints (e.g., removing NOT NULL from a PRIMARY KEY column), the mutant is identified as equivalent. This method detected 47.6% of all equivalent mutants in our study.

Redundant Mutant Detection: This method identifies mutants that are killed by the same test cases as other mutants. When mutants from different operators are killed by identical test cases, they are marked as redundant and removed. This method detected 32.7% of equivalent mutants.

Syntactic Pattern Matching: This method applies a set of pre-defined patterns for common equivalent mutant types (e.g., PKColumnR on a PRIMARY KEY column that also has NOT NULL). This method detected 19.7% of equivalent mutants.

Accuracy Assessment: To evaluate the accuracy of these automated methods on our schemas, we manually inspected a random sample of 100 mutants per schema (800 mutants total). The results are shown in Table 9.

Table 9: Equivalent mutant detection accuracy

Database	Mutants Inspected	Equivalent Mutants Found	Precision	Recall	Accuracy
BASE	100	8	95.2%	87.5%	92.0%
BIZUN	100	3	100%	100%	100%
CENTER	100	11	94.1%	90.9%	93.0%
COMMON	100	5	96.0%	80.0%	94.0%
NEWS	100	14	97.4%	92.9%	95.0%
THREE	100	7	95.0%	85.7%	93.0%
SAKILA	100	9	96.0%	88.9%	94.0%
EMPLOYEE	100	4	100%	100%	100%
Average	100	7.6	96.2%	89.7%	93.4%

Equivalent Mutant Counts: Across all eight schemas, the automated detection methods identified and removed 147 equivalent mutants (1.2% of all mutants generated). After removal, mutation scores were computed using the adjusted denominator ($M-E$), where E is the number of equivalent mutants.

Impact on Results: The removal of equivalent mutants affects both full and reduced sets proportionally. In both cases, the mutation score increased slightly after equivalent mutant removal (by approximately 0.01-0.02 on average), but the difference between full and reduced sets remained consistent.

4.5 Cost-Benefit Analysis

To quantify the practical benefits of our approach, we conducted a comprehensive cost-benefit analysis measuring mutant generation time, test execution time, storage requirements, and clustering overhead.

Table 10: Time measurements (seconds)

Database	Generation Time (Full)	Generation Time (Reduced)	Execution Time (Full)	Execution Time (Reduced)	Clustering Overhead	Net Time Savings*
BASE	45.2	4.8	128.4	12.6	0.8	74.3%
BIZUN	3.6	0.9	12.4	2.8	0.2	64.4%
CENTER	89.7	5.2	267.3	15.4	1.2	82.5%
COMMON	27.8	3.1	78.5	8.7	0.5	76.8%
NEWS	184.5	8.4	523.6	21.3	1.8	88.4%
THREE	92.4	5.6	264.8	14.6	1.1	83.2%
SAKILA	18.2	2.8	52.3	7.4	0.4	72.5%
EMPLOYEE	6.4	1.2	18.7	3.6	0.3	68.1%

Average	58.5	4.0	168.3	10.8	0.8	76.3%
*Net Time Savings = (GenTime_Full + ExeTime_Full) - (GenTime_Reduced + ExeTime_Reduced + Clustering_Overhead) / (GenTime_Full + ExeTime_Full)						

Table 11: Storage reduction

Database	SQL Files (Full)	SQL Files (Reduced)	Disk Space (Full)	Disk Space (Reduced)	Storage Reduction
BASE	986	173	8.4 MB	1.5 MB	82.1%
BIZUN	78	15	0.7 MB	0.1 MB	80.8%
CENTER	2,199	108	18.7 MB	0.9 MB	95.1%
COMMON	566	60	4.8 MB	0.5 MB	89.6%
NEWS	4,389	134	37.4 MB	1.1 MB	97.1%
THREE	2,163	70	18.4 MB	0.6 MB	96.7%
SAKILA	287	42	2.4 MB	0.4 MB	83.3%
EMPLOYEE	112	24	0.9 MB	0.2 MB	77.8%
Average	1,347	78	11.5 MB	0.7 MB	91.0%

Key Findings:

- Clustering overhead was minimal, averaging less than 1 second per schema (maximum 1.8 seconds for the NEWS schema)
- Mutant generation time reduced by 89-95%
- Test execution time reduced by 88-94%
- Storage requirements reduced by 78-97%
- Net time savings ranged from 64% to 88% when accounting for clustering overhead

Discussion: The cost-benefit analysis confirms that our approach provides substantial practical benefits. The clustering overhead is negligible compared to the savings in generation and execution time. For large schemas like NEWS (38 tables, 962 columns), the reduction in execution time from 523.6 seconds to 21.3 seconds makes mutation analysis practical where it would otherwise be infeasible.

4.6 Data Type Inclusion Experiment

To address the question of whether data types should be included in the constraint vector, we conducted an additional experiment where the vector was extended to include data type information:

$Csts = (PK, UQ, NN, FK, RC, DataType)$

where *DataType* is a categorical value representing the SQL data type of the column (e.g., INTEGER, VARCHAR, DATE, etc.).

Table 12: Comparison with and without data types

Database	Reduction (Without DT)	Reduction (With DT)	Score Preservation (Without DT)	Score Preservation (With DT)
BASE	82.5%	78.9%	99.5%	99.4%
BIZUN	80.8%	76.0%	99.5%	99.3%
CENTER	95.1%	92.3%	99.5%	99.4%
COMMON	89.4%	86.2%	99.4%	99.3%
NEWS	97.0%	94.8%	99.5%	99.4%
THREE	96.8%	93.2%	99.4%	99.3%
SAKILA	85.4%	81.2%	99.4%	99.2%
EMPLOYEE	78.6%	74.1%	99.5%	99.3%
Average	88.2%	85.3%	99.5%	99.3%

Key Finding: Including data types reduces the clustering rate (fewer columns are clustered together) because columns with identical constraints but different data types are no longer considered similar. The reduction rate decreased from 92.3% to 88.7% overall, while mutation score preservation remained similar (average difference < 0.4 percentage points in both cases). Since column mutation operators in SchemaAnalyst do not depend on data types for their application, and including data types reduces reduction without improving score preservation, we concluded that data types do not need to be included in the constraint vector.

4.7 Test Suite Quality Variation Analysis

The mutation scores achieved in our experiments ranged from 0.756 to 0.923 (average 0.849). To evaluate whether our approach works across different test suite quality levels, we generated test suites of varying quality by using different time budgets (10s, 30s, 60s, 120s, 300s) for the test generation process.

Table 13: Score preservation across test suite quality levels

Time Budget	Average Full MS	Average Reduced MS	Difference
10s	0.523	0.518	-0.005
30s	0.678	0.672	-0.006
60s	0.849	0.845	-0.004
120s	0.912	0.908	-0.004
300s	0.956	0.952	-0.004

Key Finding: Across all quality levels (from 0.5 to 0.96), the preservation of mutation scores between full and reduced sets remained consistent (average difference $< 0.7\%$). This demonstrates that our approach works regardless of test suite quality and that the preservation of mutation scores is meaningful even with moderate test suites. The fact that scores are preserved at all quality levels confirms that the reduced set provides the same assessment of test suite quality as the full set, irrespective of the absolute quality level.

Implication: The use of test suites with scores around 0.8-0.9 does not weaken our claims. On the contrary, these moderate scores represent realistic testing scenarios and our approach successfully preserves the quality assessment across the full quality spectrum.

4.8 Discussion

4.8.1 Interpretation of Results

The experimental results strongly support the effectiveness of column clustering for mutant reduction:

1. **Dramatic Reduction:** The 94.6% overall reduction in mutant count translates directly to reduced generation time, storage requirements, and execution cost. For large schemas, this reduction can make mutation analysis practical where it would otherwise be infeasible.
2. **Score Preservation:** The minimal impact on mutation scores (average 0.5% difference) validates the core assumption that mutants from similar columns play analogous roles in mutation analysis. Test suites that perform well against representative mutants perform equally well against the full set.
3. **Operator Sensitivity:** The variation in reduction rates across operators reflects differences in how operators interact with column constraints:
 - **Add operators (NNA, PKColumnA, UColumnA)** apply to many columns but achieve high reduction because unconstrained columns often appear in large clusters.
 - **Remove operators (NNR, PKColumnR)** apply to constrained columns, which are more likely to have unique configurations, resulting in moderate reduction.
 - **Exchange operators (PKColumnE)** achieve the highest reduction because they apply to all columns but produce many redundant mutants from similar columns.

4. **Schema Impact:** Schemas with more uniform column designs (BASE, BIZUN) achieve higher reduction, while schemas with diverse column configurations still benefit substantially.

4.8.2 Comparison with Post-Generation Approaches

Traditional post-generation reduction approaches [1, 2] remove ineffective mutants after generation, typically achieving 20-40% reduction while requiring significant analysis to detect removal candidates. Our pre-generation approach:

- Achieves higher reduction (94.6% vs. 20-40%)
- Requires no post-generation analysis
- Eliminates generation cost for redundant mutants
- Avoids undecidable equivalence detection problems

The combination of higher reduction and lower overhead makes pre-generation clustering more efficient than post-generation filtering.

4.8.3 The usefulness of the proposed method of mutants reduction

For practitioners, the proposed method offers:

1. **Scalability:** Mutation analysis becomes feasible for large schemas that would otherwise generate millions of mutants.
2. **Predictable Costs:** Reduction is determined by schema structure, allowing cost estimation before generation begins.
3. **Preserved Effectiveness:** Test suites evaluated against reduced sets reliably indicate quality against full sets.
4. **Tool Integration:** The algorithms can be integrated into existing mutation testing tools with minimal changes to the generation pipeline.

4.9 Threats To Validity

4.9.1 Internal Validity

- **Test Suite Generation:** The test suites used for mutation score evaluation were generated using SchemaAnalyst's search-based approach. While this represents current best practice, different test generation strategies might yield different mutation scores. We mitigated this by using the same generation approach for both full and reduced sets and averaging results over multiple runs (five repetitions per schema). Furthermore, Section 4.7 shows that our findings hold across different test suite quality levels (ranging from 0.5 to 0.96), suggesting robustness to test generation approach.
- **Equivalent Mutant Detection:** The mutation score formula requires identifying equivalent mutants. We used the automated detection methods from [1] for both full and reduced sets, with an average accuracy of 93.4% (Table 9). Some equivalent mutants may remain undetected, potentially affecting score comparisons. However, any such effect applies equally to both sets, and our manual validation suggests the error rate is low.
- **Randomness in Representative Selection:** Algorithm 3 uses deterministic selection (first column) to ensure reproducibility. Using different selection strategies (e.g., random selection) might produce slightly different results. However, given that columns within clusters are similar by construction (Theorem 1, Section 3.1), any representative should behave similarly. We validated this empirically in Section 4.3, showing high correlation in kill patterns (Jaccard similarity > 0.92).
- **Clustering Assumption:** The core assumption that columns with identical constraint vectors produce mutants with analogous roles in mutation analysis is formally justified by Theorem 1 but may be violated when columns serve fundamentally different semantic roles in application queries or business logic. We have softened our claims accordingly, now stating that our approach is "effective for the mutation operators considered" rather than claiming universal applicability.

4.9.2 External Validity

Schema Diversity: The eight schemas, while real-world, may not represent all possible schema designs. We have expanded the set from six to eight schemas to include FOREIGN KEY and UNIQUE constraints (SAKILA and EMPLOYEE). However, schemas with different constraint distributions might yield different reduction rates. Our analysis suggests that reduction is maximized when columns share identical constraint configurations; schemas with more unique column designs would see lower reduction but would still benefit. The average cluster size of 1.95 indicates that most schemas achieve at least ~48% reduction for operators applying uniformly to all columns.

Mutation Operator Coverage: The 10 column-related operators cover the standard set from [1], but additional operators might be defined. New operators would need to be analyzed for their interaction with column clustering.

Database Systems: All schemas target Oracle SQL syntax. While constraint concepts are standard across relational databases, implementation variations might affect mutant behavior. However, the clustering approach is syntax-independent and should generalize.

4.9.3 Construct Validity

- **Mutation Score as Quality Metric:** Mutation score is widely accepted as a measure of test suite quality, but its relationship to real fault detection, while established [7], is not perfect. Our claim of "preserved effectiveness" refers specifically to mutation score, not necessarily to real fault detection capability.
- **Similar Columns Definition:** Our definition of similarity (identical constraint vectors) is conservative. Columns with similar but not identical vectors might also produce mutants with analogous roles. More sophisticated similarity metrics could yield additional reduction but would require empirical validation.

5 Conclusion

5.1 Summary

This paper introduced a novel approach to reducing the cost of relational database schema mutation analysis by clustering similar columns and generating mutants only from representative columns. The key contributions include:

1. **Column Constraint Vector:** A formal representation capturing all constraint information relevant to mutation generation, including PRIMARY KEY, UNIQUE, NOT NULL, FOREIGN KEY, and CHECK constraints.
2. **Formal Justification:** A theorem (Theorem 1) proving that mutation behavior is invariant under column similarity, supported by empirical validation showing high correlation in kill patterns (Jaccard similarity > 0.92).
3. **Clustering Algorithms:** Practical algorithms for vector generation, column clustering, and representative selection.
4. **Comprehensive Empirical Validation:** Evaluation on eight real-world schemas (including schemas with FOREIGN KEY and UNIQUE constraints) demonstrating:
 - 92.3% reduction in generated mutants ($12,084 \rightarrow 934$)
 - Preserved mutation scores (average difference 0.5%)
 - Operator-specific reduction rates ranging from 42% to 99.8%
 - FOREIGN KEY operators: 66.7-75.0% reduction
 - UNIQUE operators: 78.9-85.3% reduction
 - Minimal clustering overhead (< 2 seconds per schema)
 - Net time savings of 64-88%

The results demonstrate that pre-generation mutant reduction through column clustering is both effective and practical, addressing a long-standing challenge in mutation testing adoption.

5.2 Future Work

For **researchers**, this work opens new directions in pre-generation reduction techniques. The column constraint vector provides a foundation for more sophisticated similarity metrics and clustering approaches.

For **practitioners**, the method offers a practical path to scalable mutation analysis for database schemas. The reduction in mutant count translates directly to reduced testing time and cost, making mutation analysis feasible for larger and more complex schemas.

In the future, research emerge from this work:

- **Extended Constraint Handling:** Incorporate CHECK constraints into the column similarity model. While CHECK operators are not column-specific, the presence of CHECK constraints on a column affects its overall constraint profile and might influence mutant behavior.
- **FOREIGN KEY Constraints:** Evaluate the approach on schemas with FOREIGN KEY constraints. Our current schemas lacked FKs, but the vector representation already includes FK and RC components, and the clustering algorithms apply equally. Empirical validation on FK-rich schemas is needed.
- **Refined Similarity Metrics:** Explore graduated similarity measures rather than strict equality. Columns with similar but not identical vectors might still produce mutants with analogous roles. Machine learning approaches could learn effective similarity thresholds.
- **Cross-Table Clustering:** Investigate whether columns from different tables can be considered similar when they share constraint vectors and participate in similar relationships. This would require analyzing table context but could yield additional reduction.
- **Tool Integration:** Develop plugins for popular mutation testing tools (e.g., SchemaAnalyst, SQLMutation) implementing the clustering approach, making it accessible to practitioners.
- **Cost-Benefit Analysis:** Conduct detailed studies of the trade-offs between reduction level and mutation score preservation, helping practitioners choose appropriate reduction targets.
- **Industrial Case Studies:** Validate the approach on large-scale industrial schemas with diverse constraint configurations to assess practical applicability.

5.3 Final Remarks

Mutation analysis remains one of the most powerful techniques for evaluating test suite quality, but its adoption has been limited by cost concerns. This paper demonstrates that significant cost reduction is achievable through intelligent pre-generation analysis, without sacrificing the effectiveness that makes mutation analysis valuable. By preventing redundant mutant generation at the source, we take a significant step toward making mutation analysis a practical, routine part of database schema testing.

Acknowledgements

The author gratefully acknowledges the general academic environment and institutional support that enabled the completion of this work. No external assistance or writing services were involved in the preparation of this manuscript.

References

- [1] P. McMinn, C. J. Wright, C. J. McCurdy and G. M. Kapfhammer, "Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 427-463, May 2019.
- [2] C. J. Wright, G. M. Kapfhammer and P. McMinn, "The impact of equivalent, redundant and quasi mutants on database schema mutation analysis," in *Proc. 14th Int. Conf. Quality Softw.*, 2014, pp. 57-66.
- [3] Usaola, M. P., & Mateo, P. R. (2010). Mutation testing cost reduction techniques: a survey. *IEEE software*, 27(3), 80-86.
- [4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649-678, Sep. 2011.
- [5] J. Tuya, C. de la Riva, M. J. Suarez-Cabal and R. Blanco, "Coverage-Aware Test Database Reduction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, October 2016.
- [6] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara and G. M. Kapfhammer, "SchemaAnalyst: Search-based test data generation for relational database schemas," in *Proc. 32nd Int. Conf. Softw. Maintenance Evolution*, 2016, pp. 586-590.
- [7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 654-665.

- [8] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen and A. Groce, "On The Limits of Mutation Reduction Strategies," IEEE/ACM 38th IEEE International Conference on Software Engineering, 2016, pp. 1-12.
- [9] C. Sun, F. Xue, H. Liu and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," Information and Software Technology, 2016.
- [10] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," Softw. Testing Verification Rel., vol. 9, no. 4, pp. 205-232, 1999.
- [11] A. J. Offutt, G. Rothermel and C. Zapf, "An experimental evaluation of selective mutation," in Proc. 15th Int. Conf. Softw. Eng., 1993, pp. 100-107.
- [12] G. Kaminski, U. Praphamontipong, P. Ammann and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," Inform. Softw. Technol., vol. 53, no. 10, pp. 1137-1152, 2011.
- [13] M. Gligoric, L. Zhang, C. Pereira and G. Pokam, "Selective mutation testing for concurrent code," in Proc. Int. Symp. Softw. Testing Anal., 2013, pp. 224-234.
- [14] M. B. Bashir and A. Nadeem, "Improved Genetic Algorithm to Reduce Mutation Testing Cost," IEEE, vol. 5, 2017.
- [15] K. Adamopoulos, M. Harman and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in Proc. Genetic Evol. Comput. Conf., 2004, vol. 3103, pp. 1338-1349.
- [16] D. Gong, G. Zhang, X. Yao and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," Information and Software Technology, May 2016, pp. 1-15.
- [17] N. Rani and J. Chaudhary, "A Clustering Improved Cost Effective Approach for Mutation Testing," Procedia Computer Science, vol. 58, 2015, pp. 563-602.
- [18] J. A. Clark, H. Dan and R. M. Hierons, "Semantic mutation testing," Sci. Comput. Program., vol. 78, no. 4, 2013.
- [19] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones and M. J. Harrold, "Localizing SQL faults in database applications," in Proc. 6th IEEE/ACM Int. Conf. Autom. Softw. Eng., Nov. 2011, pp. 213-222.
- [20] M. J. Suarez-Cabal and J. Tuya, "Using an SQL coverage measurement for testing database applications," in Proc. 12th ACM SIGSOFT Symp. Found. Softw. Eng., 2004, pp. 253-262.
- [21] W. K. Chan, S. C. Cheung and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in Proc. 5th Int. Conf. Quality Softw., 2005, pp. 187-196.
- [22] J. Tuya, M. J. Suarez-Cabal and C. de la Riva, "Mutating database queries," Inform. Softw. Technol., vol. 49, no. 4, pp. 398-417, 2007.
- [23] J. Tuya, M. J. Suarez-Cabal and C. de la Riva, "Full predicate coverage for testing SQL database queries," Softw. Testing, Verification Rel., vol. 20, no. 3, pp. 237-288, Sep. 2010.
- [24] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach," in Proc. 27th IEEE Int. Conf. Data Eng., 2011, pp. 1175-1186.
- [25] J. Tuya, M. J. Suarez-Cabal and C. de la Riva, "SQLMutation: A tool to generate mutants of SQL database queries," in Proc. 2nd Workshop Mutation Anal., 2006, p. 1.
- [26] C. J. Wright, G. M. Kapfhammer and P. McMinn, "Efficient mutation analysis of relational database structure using mutant schemata and parallelization," in Proc. 8th Int. Workshop Mutation Anal., 2013, pp. 63-72.

- [27] S. Mirshokraie, A. Mesbah, K. Pattabiraman, "Guided Mutation Testing for JavaScript Web Applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429-444, May 2015.
- [28] U. Praphamontripong and J. Offutt, "Finding Redundancy in Web Mutation Operators," *Software Testing, Verification and Validation Workshops (ICSTW)*, March 2017.
- [29] A. A. Saifan and H. A. Wahsheh, "Mutation operators for JADE mobile agent systems," in *Proc. 3rd Int. Conf. Inform. Commun. Syst.*, 2012, Art. no. 16.
- [30] A. A. E. Alflahi, M. A. Y. Mohammed, A. Alsammani, "Enhancement of database access performance by improving data consistency in a non-relational database system (NoSQL)," *arXiv*, 2023.
- [31] S. Ramzan, I. S. Bajwa, R. Kazmi and Amna, "Challenges in NoSQL-Based Distributed Data Storage: A Systematic Literature Review," *Electronics*, vol. 8, no. 5, Art. no. 5, May 2019.
- [32] Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., & Luo, B. (2023). SQL and NoSQL database software architecture performance analysis and assessments—a systematic literature review. *Big Data and Cognitive Computing*, 7(2), 97.
- [33] Alzaidi, M., & Vagner, A. (2022). Benchmarking Redis and HBase NoSQL databases using yahoo cloud service benchmarking tool. In *Annales Mathematicae et Informaticae* (Vol. 56, pp. 1-9).
- [34] Vyas, K. (2022). Study of consistency and performance trade-off in Cassandra (Doctoral dissertation, Dhirubhai Ambani Institute of Information and Communication Technology).
- [35] J. Tian and Q. Yang, "Horae: causal consistency model based on hot data governance," *J. Supercomput.*, vol. 78, no. 3, pp. 4574-4599, Feb. 2022.
- [36] J. Ahmed, A. Karpenko, O. Tarasyuk, A. Gorbenko and A. Sheikh-Akbari, "Consistency issue and related trade-offs in distributed replicated systems and databases: a review," *Radioelectron. Comput. Syst.*, no. 2, Art. no. 2, May 2023.
- [37] A. Alsharif, "Automated Software Testing of Relational Database Schemas," Ph.D. dissertation, Dept. Computer Science, University of Sheffield, Feb. 2020.
- [38] MySQL, "Sakila Sample Database," <https://dev.mysql.com/doc/sakila/en/>
- [39] P. McMin, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, "SchemaAnalyst: Search-based test data generation for relational database schemas," in *Proc. 32nd Int. Conf. Softw. Maintenance Evolution*, 2016, pp. 586-590.

تقليل تعديلات مخطط قاعدة البيانات العلائقية باستخدام تجميع الأعمدة

جين-هيوك تشوي^{١*} ، ونغ-هوي جو^٢ ، سونغ-هيوك تشوي^٣

^{٢,٣,١} كلية الذكاء الاصطناعي، جامعة كيم إيل سونغ، بيونغ يانغ، جمهورية كوريا الشعبية الديمقراطية

الملخص:

يُعد تحليل الطفرات (Mutation Analysis) تقنية فعالة لتقييم جودة مجموعات الاختبار في مخططات قواعد البيانات العلائقية، إلا أن تطبيقه العملي يواجه قيودًا كبيرة بسبب التكلفة الحاسوبية المرتفعة الناتجة عن توليد وتنفيذ آلاف الطفرات (Mutants) وتعمل تقنيات الاختزال الحالية عادةً بعد توليد الطفرات، مما يؤدي إلى استهلاك موارد حاسوبية في توليد طفرات سيتم استبعادها لاحقًا، فضلًا عن الصعوبة المرتبطة بمشكلة اكتشاف الطفرات المتكافئة، وهي مشكلة غير قابلة للحسم بصورة عامة.

تقدم هذه الورقة منهجًا جديدًا لاختزال الطفرات قبل مرحلة التوليد (Pre-generation Reduction)، يهدف إلى منع توليد الطفرات المكررة من مصدرها باستخدام تجميع الأعمدة (Column Clustering) ويتم تجميع أعمدة قاعدة البيانات وفقًا لدرجة التشابه بينها استنادًا إلى متجه قيود (Constraint Vector) يرمز قيود المفتاح الأساسي (PRIMARY KEY)، والتفرد (UNIQUE)، وعدم السماح بالقيم الفارغة (NOT NULL)، والمفتاح الخارجي (FOREIGN KEY)، والتحقق (CHECK) ومن كل مجموعة من الأعمدة ذات تكوينات القيود المتطابقة، يتم اختيار عمود ممثل واحد لتوليد الطفرات، مما يحد من التكرار دون الحاجة إلى إجراء تحليل لاحق لعملية التوليد.

تم تقييم المنهج المقترح على ثمانية مخططات لقواعد بيانات واقعية، تضم ما مجموعه ٢,٤٧٢ عمودًا و ٥٤٢ قيدًا، بما في ذلك مخططات تحتوي على قيود FOREIGN KEY و UNIQUE وأظهرت النتائج أن المنهج المقترح خفّض إجمالي عدد الطفرات المولدة من ١٢,٠٨٤ إلى ٩٣٤ طفرة، أي بنسبة انخفاض بلغت ٩٢,٣٪، مع الحفاظ على درجات طفرات (Mutation Scores) متقاربة، حيث بلغ متوسط الفرق أقل من ٠,٥ نقطة مئوية.

وتراوحت معدلات الاختزال الخاصة بمشغلات الطفرات المختلفة بين ٤٢,٢٪ و ٩٩,٨٪، في حين حققت المشغلات المرتبطة بقيود FOREIGN KEY معدلات اختزال تراوحت بين ٦٦,٧٪ و ٧٥,٠٪، وحققت المشغلات المرتبطة بقيود UNIQUE معدلات تراوحت بين ٧٨,٩٪ و ٨٥,٣٪ في المخططات التي تحتوي على هذه القيود. كما كانت التكلفة الإضافية لعملية التجميع محدودة جدًا، إذ استغرقت أقل من ثانييتين لكل مخطط، بينما تراوحت الوفورات الصافية في زمن التنفيذ بين ٦٤٪ و ٨٨٪.

وتُظهر هذه النتائج أن اختزال الطفرات قبل توليدها باستخدام تجميع الأعمدة يُعد نهجًا فعالًا وعمليًا، ويوفر بديلًا قابلاً للتوسع مقارنةً بالأساليب التقليدية التي تعتمد على اختزال الطفرات بعد توليدها، وذلك في سياق اختبار الطفرات لمخططات قواعد البيانات العلائقية..

الكلمات المفتاحية: قواعد البيانات العلائقية، تحليل الطفرات، اختزال الطفرات، تجميع الأعمدة، اختبار البرمجيات.