Generating Test Data to Distinguish Conjunctive Queries with Equalities

Preetham Vemasani, Alexander Brodsky, Paul Ammann
Department of Computer Science
George Mason University
Fairfax, VA, USA
Email: pvemasan@masonlive.gmu.edu, brodsky@gmu.edu, pammann@gmu.edu

***Abstract*—The widespread use of databases in software systems has increased the importance of unit testing the queries that form the interface to these databases. Mutation analysis is a powerful testing technique that has been adapted to test database queries. But each of the existing mutation approaches to testing database queries has one or more of the following shortcomings: inability to recognize equivalent mutants, inability to generate test databases automatically, or inability to mutate all aspects of a query. In this paper we address all three of these challenges by adapting results from the rich literature on query rewriting. We restrict attention to the class of conjunctive queries with equalities. In return for this restriction, we give an algorithm that recognizes equivalent mutants, generates a test database that distinguishes each nonequivalent mutant, and applies to arbitrary mutations, as long at the mutation is also a conjunctive query with equalities. The paper presents the test database generation algorithm and proves that it is sound and complete for conjunctive queries with equalities. We then illustrate the algorithm on a sample query. We evaluate mutations of the query both with the new technique and compare the results to existing mutation techniques for databases.**

***Keywords*-Mutation testing, Database Query Testing**

I. Introduction

Queries form the interface between software systems and databases. To assure high quality in such systems, queries need to be evaluated to determine if they capture correctly the designer's intent. Queries are often evaluated by testing, which means executing them on some instances of the database and judging the results. Testing is especially important for database queries because more formal approaches to verification are sometimes hard to apply: the query may well be the first description that formally captures the designer's intent. Many commercial tools focus on database performance instead of functional behavior, but the gap in assessing correct behavior, first identified by Davies *et al* [1], is still significant. A variety of subsequent research efforts (see related work) can be used to populate the test databases with data appropriate for evaluating correctness.

One powerful approach to generating test data is mutation analysis [2]. Mutation analysis produces variants of an artifact, known as mutants, typically, using a set of predefined rules known as mutation operators. A test set classifies mutants as either killed (the mutant computes a different outcome than the artifact for at least one test in the test set) or live (not killed). Live mutants may be *equivalent*, which means they compute the same function for all possible tests. In general, determining if a mutant is equivalent is undecidable, but for some domains, including the one considered in this paper, the problem does indeed become decidable.

Mutation analysis has been used both to test databases directly, and also to evaluate the effectiveness of other approaches. But existing approaches to testing database queries with mutation analysis have one or more of the following shortcomings: an inability to recognize equivalent mutants, an inability to generate test databases automatically, or an inability to mutate all aspects of a query.

In general, many mutation operators produce a large number of mutants and may also generate equivalent mutants. An ideal testing technique not only filters out the equivalent mutants, but also generates minimal test data to kill the non-equivalent mutants. This paper explores the idea of generating test data by leveraging query containment results for a particular class of queries. The goal is to generate minimal test data while eliminating equivalent mutants – and to do so for *arbitrary* mutants inside the particular class of queries.

Although verifying query containment in general is undecidable [3], extensive research work in the field of query optimization has shown that it is not the case with many useful classes of database queries. Containment of database queries over relational databases has been widely studied for various classes of queries such as conjunctive queries [3]; conjunctive queries with comparisons [4]; aggregate queries [5], etc. As equivalence between two queries can be viewed as containment in both directions, the problem of equivalent mutants can also be addressed by this approach.

In this paper, we propose a technique to generate test data for conjunctive queries with equalities. This technique uses query containment properties to either generate an instance of a database which distinguishes a query from a mutation (in the case that containment does not hold in at least one direction) or declare that the mutation is equivalent to the query (in the case that containment holds in both directions). The technique applies to arbitrary mutants, as long as the mutants are also conjunctive queries with equalities. In addition, redundant mutants can be eliminated by using the transitive property of the query containment relation.

The contributions of this paper are:

- A mutation-based test data generation algorithm based

upon query containment for the class of conjunctive queries with equalities. The technique applies to arbitrary mutants inside the class of conjunctive queries with equalities.
- A mathematical proof of soundness and completeness for the algorithm. That is, the algorithm always generates a test database if possible for a given mutant, and always declares the mutant equivalent if not.
- An evaluation of the technique on a simple SQL query and all mutants generated from it using the SQLMutation tool [6].

The paper is organized as follows. The relation of this work to prior test data generation for databases is presented in section II. As this technique is based on relational conjunctive queries, the process of converting SQL queries to relational queries is described in section III. The test data generation algorithm and accompanying mathematical proof are presented in section IV. The evaluation is presented in section V. Conclusions and future work are presented in section VI.

II. Previous Work

In this section, we briefly describe various testing techniques proposed for database queries and in particular focus on mutation based techniques.

Davies *et al* [1] first identified the gap in testing functional correctness of databases. AGENDA [7] is the first research prototype which articulated the basic issues in testing database-driven applications. It helped the test engineers to produce test data tied to the semantic constraints of the application. Advancements in this approach include a semi automated approach [8] where test engineers state the desired properties of the test database instead of manually generating them. Another semi automatic testing approach is Reverse Query Processing [9] which starts with an expected output. Queries are then analyzed to produce a database instance (serves as test data) which will generate the given expected output. The quality of the test database in this approach depends on the expected output provided.

Some techniques have been proposed using various coverage criteria for populating the test database with significant data. [10] proposed combinatorial coverage of conditions in SQL predicates and data-flow criteria was used for testing application database interactions in the work of [11]. The cost of the combinatorial coverage approach increases exponentially with the increase in the number of conditions. In ordinary programs, mutation based tests have been found to be stronger than the data flow tests. The coverage criterion in [12] was SQLFpc, which is equivalent to Multiple Condition/Decision Coverage (MCDC). Although MCDC only guarantees detection of 2 of the 9 faults in the Lau and Yu fault hierarchy [13], it is still viewed as a strong criterion in practice. Tuya *et al* [14] proposed a set of mutation operators tailored for SQL queries and a tool called SQLMutation [6] which implements these operators.

Kaminski and Ammann [15] proposed higher order TRF/TIF mutation operator to address these issues for the Lau and Yu fault hierarchy [13]. TRF/TIF mutation operators produced fewer equivalent mutants and test databases generated using XOR method produced competitive mutation scores with respect to SQLMutation. Though these results are encouraging, they are confined to the Lau and Yu fault hierarchy and limited to SELECT queries having disjunctive normal form (DNF) expressions in their WHERE clauses [16].

Pan *et al* [17]. suggested a new approach called MutaGen (Test Generation for Mutation Testing on Database Applications) to optimize mutation testing for database queries. This approach treats a query as a program and uses the SynDB framework, based on dynamic symbolic execution, to generate test databases. Empirical analysis of this approach shows promising results, but the technique does not kill all mutants. Shah *et al* [18] proposed an algorithm which identifies constraints that need to be satisfied by the test dataset to kill a particular group of mutants. Though this technique identifies all the equivalent mutants and kills all the non-equivalent mutants, these results are confined to a particular set of mutation operators. Search based test data generation techniques presented in [19] and [20] achieved better mutation scores in comparison with the popular data generation tools such as *DBMonster*. Unlike the data generation algorithm given in this paper, the focus of this data generation technique is testing the integrity constraints of the database schema.

The relation of the current work to all of these efforts basically boils down to trading scope – namely, the restriction to conjunctive queries with equalities – for effectiveness within that scope – namely, the ability to identify all equivalent mutants and generate data to distinguish all non-equivalent mutants, so long as they are in the proper class of queries.

III. Conjunctive Queries

In this section, we describe the semantics of Conjunctive queries. Since our motivation is the test data generation for real-life queries, we start with the description of SQL Conjunctive queries.

A. SQL Conjunctive Queries

The basic form of an SQL query as described in [21] is as follows:

```
SELECT DISTINCT select-list
FROM   from-list
WHERE  qualification
```

where, `select-list` is the list of relation attributes to be retained in the result, `from-list` is the list of relations(tables), `qualification` is the conjunction of comparisons among relation attributes (only conjunctive queries with equalities are considered in this paper) and `DISTINCT` eliminates duplicate tuples in the result.

Example 1: The following is an example of a Conjunctive query which extracts the name and avg_rating of Textbooks whose cost is equal to 100$ and their average rating is equal to the critic rating for the book.

```
SELECT DISTINCT t.name, t.rating
```

```
FROM Textbooks t, Reviews r
WHERE  t.id = r.ID
AND t.cost = 100
AND t.rating = r.critic
```

Here, `Textbooks` and `Reviews` are relations in the database with attributes `(id, name, cost, rating)` and `(id, user, critic)` respectively. `Textbooks` contains the details of all the books and the averages of the user and critic ratings given to the respective book are stored in the relation `Reviews`.

The operational semantics of SQL conjunctive queries can be defined as follows (refer to [21] for complete details):

1) The cross product of the relations in `from-list` is computed.
2) Tuples which do not confine to the `qualification` conditions are removed.
3) Columns mentioned in the `select-list` are retained.

Example 2: The following example depicts the sequence of steps followed during the execution of an SQL query given in Example 1.

Let an instance of the relation `Textbooks` be

id	name	cost	rating
1024	DBMS	100	4.3
2048	Testing	100	4.4

and that of `Reviews` be

id	user	critic
1024	4.3	4.3
2048	4.9	4.2

Therefore, the cross product of the relations results in:

Textbooks				Reviews		
id	**name**	**cost**	**rating**	**id**	**user**	**critic**
1024	DBMS	100	4.3	1024	4.3	4.3
1024	DBMS	100	4.3	2048	4.9	4.2
2048	Testing	100	4.4	1024	4.3	4.3
2048	Testing	100	4.4	2048	4.9	4.2

After applying the conditions in the $qualification$, only the following tuples remain:

Textbooks				Reviews		
id	**name**	**cost**	**rating**	**id**	**user**	**critic**
1024	DBMS	100	4.3	1024	4.9	4.3

Finally by retaining the columns specified in the *select-list*, we obtain the result of query in Example 1 as:

t.name	t.rating
DBMS	4.3

B. Relational Conjunctive Queries

In this subsection, we describe the relational syntax of conjunctive queries with equalities and their semantics as in [3] and [22].

Let D be a relational database. A conjunctive query with equalities over D is an expression of the form,

$$Q(\bar{Z}) \leftarrow R_1(\bar{X_1}) \wedge ..... \wedge R_n(\bar{X_n})$$

where $n > 0$ and $R_1, ...R_n$ are relation names in D. Q is a relation name not in D and its tuples form the result of the query. $Q(\bar{Z})$ is the head of the query and $R_1(\bar{X_1}), ..., R_n(\bar{X_n})$ is the body of the query. $\bar{Z}$, $\bar{X_1}, ..., \bar{X_n}$ are free tuples which may contain variables or constants. Arity of $\bar{X_i}$ is the arity of R_i. Each variable in $\bar{Z}$ (known as *selected variables*) must occur at least once in $\bar{X_1}, ..., \bar{X_n}$ and the variables, other than those of $\bar{Z}$ in the query, are known as *distinguished variables*. Note that there are no explicit equalities in this representation, rather, equalities are captured by multiple occurrences of the variables.

For a relational database D, let S represent all the symbols(variables and constants) in the query and Dom(D) represent the domain of the query. The result of running a query Q on database D is the set of tuples from the assignment mapping over D.

$Ans(Q, D) = \{m(\bar{Z}) \mid m : S \rightarrow Dom(D)$ *is an assignment mapping for* $Q\}$

An assignment mapping 'm' is the assignment of data values in D to the variables of Q such that,

- $(\forall c \in S) m(c) = c$, where c is a constant
- $(\forall i, 1 \leq i \leq n) m(X_i) \in r_i$, where r_i is a relational instance over R_i

C. Relational vs. SQL Conjunctive Queries

In this section, we briefly sketch a transformation from the SQL syntax to relational syntax of the conjunctive queries(Refer to [22] for complete details).

We will use the canonical form of SQL query as introduced in the beginning of Section III. For simplicity, let us assume all the queries under consideration are consistent i.e., all the variables that appear in the head of the query are also present in the body of the query and safe i.e., no two different constants are implied to be equal. Let $S_{\equiv}$ represent all the equivalent classes[1] present in the body of the query generated using the `qualification` conditions. The following sequence of steps generate the relational syntax.

1) For every attribute of every relation in the `from-list`, introduce a distinct variable.
2) For every relation in `from-list` introduce a corresponding conjunct with the variables in the same order as the attributes in the schema information.
3) For every equivalent class $E \in S_{\equiv}$
   - if there is a constant c in E, replace all the variables in E with c. (As the queries are assumed to be safe, there can only be at most one constant in E)
   - otherwise, replace all the variables in E with some variable in E

Example 3: The resultant relational query for the SQL query in Example 1 will be

Query(t.name, t.rating) $\leftarrow$ *Textbooks(t.id, t.name, t.cost, t.rating)* $\wedge$ *Reviews(r.id, r.user, r.critic)* $\wedge$ *t.id = r.id* $\wedge$ *t.cost = 100* $\wedge$ *t.rating = r.critic*

[1]An equivalent class of S with respect to an equivalence relation '=' is a maximal subset of S in which all elements are equivalent.

which implies,

Query(t.name, t.rating) $\leftarrow$ *Textbooks(t.id, t.name, 100, t.rating)* $\wedge$ *Reviews(t.id, r.user, t.rating)*

The results obtained by the original query and the transformed query over any relational database D will be equal[22].

IV. Generating test data

According to [3], for any two conjunctive queries with equalities say Q and $Q^{|}$, if Q is contained in $Q^{|}$, then there exists a homomorphism from $Q^{|}$ to Q. A homomorphism is a substitution of the variables of one query ($Q^{|}$) with the terms of another query (Q) and it is the basic idea behind this algorithm.

A. Algorithm to generate test database

Test database D_T is generated as follows:

1) Choose different representative symbols for all the variables present in the query.
2) For every relational atom $R_i(s_1, ..., s_m)$ in the body of the query, add a tuple $(s_1, ..., s_m)$ to the corresponding relation of R_i. Here, we use s_j as the representative symbol of itself i.e., $(\forall j, 1 \leq j \leq m) s_j$ is a representative symbol of s_j.

Note that constant itself is a representative symbol for a constant.

The database instance constructed using the above mentioned technique will just satisfy the necessary constraints in a query and hence will distinguish any other query which is not contained within itself. This algorithm populates the relations present in the query only. So, the cost of such data generation will be in linear with the execution time of the query. A mathematical proof for this claim is presented in the following sub-section.

B. Proof of soundness and completeness

Claim: Let Q_1, Q_2 be two relational conjunctive queries and D_T be a test database constructed using Q_1 then the following holds:

$$Ans(Q_1, D_T) \subseteq Ans(Q_2, D_T) \Leftrightarrow Q_1 \subseteq Q_2$$

Proof:

- '$\Leftarrow$' direction: We need to show that

  $$Ans(Q_1, D_T) \subseteq Ans(Q_2, D_T) \Leftarrow Q_1 \subseteq Q_2$$

  If $Q_1 \subseteq Q_2$, then by definition, for every database D,

  $$Ans(Q_1, D) \subseteq Ans(Q_2, D)$$

  In particular this holds for database D_T i.e.,

  $$Ans(Q_1, D_T) \subseteq Ans(Q_2, D_T)$$

- '$\Rightarrow$' direction: We need to show that

  $$Ans(Q_1, D_T) \subseteq Ans(Q_2, D_T) \Rightarrow Q_1 \subseteq Q_2$$

  Let Q_1 and Q_2 be of the form:
  - $Q_1(\bar{Z}) \leftarrow R_1(\bar{X_1}) \wedge R_2(\bar{X_2}) \wedge ..... \wedge R_n(\bar{X_n})$
  - $Q_2(\bar{W}) \leftarrow R'_1(\bar{Y_1}) \wedge R'_2(\bar{Y_2}) \wedge ..... \wedge R'_m(\bar{Y_m})$

  and let $\bar{X}$ and $\bar{Y}$ be the set of symbols (both variables and constants) of Q_1 and Q_2 respectively i.e., $\bar{X} = \bar{X_1} \cup ..... \cup \bar{X_n}$ and $\bar{Y} = \bar{Y_1} \cup ..... \cup \bar{Y_m}$.
  From the construction of D_T, it follows that

  $$\bar{Z} \in Ans(Q_1, D_T)$$

  As $Ans(Q_1, D_T) \subseteq Ans(Q_2, D_T)$,

  $$\bar{Z} \in Ans(Q_2, D_T)$$

  Note that by definition of D_T, Dom(D_T) = $\bar{X}$. Therefore, there exists an assignment mapping $m : \bar{Y} \rightarrow \bar{X}$ of Q_2, such that

  $$\bar{Z} = m(\bar{W}) \quad (1)$$

  Now, we need to show $Q_1 \subseteq Q_2$ i.e., for every database D,

  $$Ans(Q_1, D) \subseteq Ans(Q_2, D)$$

  Assume that, $\bar{Z_o} \in Ans(Q_1, D)$. Therefore, there exists an assignment mapping $m_1 : \bar{X} \rightarrow Dom(D)$ such that,

  $$\bar{Z_o} = m_1(\bar{Z}) \quad (2)$$

  To show that $\bar{Z_o} \in Ans(Q_2, D)$, we need to show that there exists an assignment mapping $m_2 : \bar{Y} \rightarrow Dom(D)$ such that $\bar{Z_o} = m_2(\bar{W})$

  Let, $m_2 = m \circ m_1$ be the composition of $m : \bar{Y} \rightarrow \bar{X}$ and $m_1 : \bar{X} \rightarrow Dom(D)$. We now show that m_2 as defined, is an assignment mapping for Q_2 such that $m_2(\bar{W}) = \bar{Z_o}$. Indeed,

  $$\begin{aligned} m_2(\bar{W}) &= (m \circ m_1)(\bar{W}) \\ &= m_1(m(\bar{W})) \\ &= m_1(\bar{Z}) && from(1) \\ &= \bar{Z_o} && from(2) \end{aligned}$$

Hence the proof of the claim.

V. Evaluation

The SQLMutation tool [6] implemented four categories of mutation operators, namely, IR (Identifier Replacement) – operators which replace columns etc; NL (Nulls) – operators handling NULL values; OR (Operator Replacement) – operators which modify expressions; SC (SQL Clauses) – operators performing mutations on main clauses etc. This mutation tool is used to generate mutants for the SQL query in Example 1 producing a total of 109 mutants. Out of these mutants, 17 mutants do not belong to the class of conjunctive queries with equalities and another 18 mutants used special functions such as 'COALESCE' which are not covered in the context of this paper. As any disjunctive queries can be written as an union of conjunctive queries, disjunctive mutants are considered as conjunctive queries in this evaluation. So, the database generation technique proposed in previous subsections is used to kill the remaining 74 mutants.

The following process is used to generate the necessary database instances:

1) Using the database generation algorithm, generate the database instance for the original query.
2) Mutants (which are not killed yet) are executed on the newly generated database instance and declared as killed if it produces a different result than the original query.
3) If any mutants are not killed, select one of the mutants which is not killed to generate a new database instance using the proposed technique.
4) If the selected mutant and original query produces the same result then, the mutant is declared as equivalent mutant and go to Step-3; else, go to Step-2.

Note that each representative symbol in the database instance is replaced with an appropriate unique constant which is not present in any of the mutants or the original query. For example, the test database instance generated for the original query is

Textbooks			
id	**name**	**cost**	**rating**
S_{id}	S_{name}	100	S_{rating}

Reviews		
id	**user**	**critic**
S_{rid}	S_{user}	S_{rating}

If we replace each symbolic constant with an appropriate value, we obtain the test database that kills the mutant query given above:

Textbooks			
id	**name**	**cost**	**rating**
1	Software Testing	100	4.22

Reviews		
id	**user**	**critic**
1	3.45	4.22

This test database can also kill a number of other mutants. To find out, we ran it against all 74 mutants in the conjunctive query with equalities class. The result was that it killed 54 of these mutants. We then selected one of the 20 mutants remaining, and reapplied our technique. In this case, we selected a mutant generated by the NULL mutation operator:

```
SELECT DISTINCT t.name, t.rating
FROM Textbooks t, Reviews r
WHERE  t.id = r.id
AND (t.cost IS NULL)
AND t.rating = r.critic
```

which produced the database instance:

Textbooks			
id	**name**	**cost**	**rating**
2	Data Mining	NULL	4.23

Reviews		
id	**user**	**critic**
2	3.46	4.23

This resulting database killed 6 of these 20 mutants. The process continued until only 2 mutants remained. Applying our technique to these 2 mutants produced database instances which are similar to the instance generated by the original query. As these are new database instances, the representative symbols are replaced with different values. For example, equivalent mutant generated by the IR (Identity Replacement) mutation operator is:

```
SELECT DISTINCT t.name, r.critic
FROM Textbooks t, Reviews r
WHERE  t.id = r.id
AND t.cost = 100
AND t.rating = r.critic
```

which produced the database instance:

Textbooks			
id	**name**	**cost**	**rating**
7	Compilers	100	4.27

Reviews		
id	**user**	**critic**
7	3.50	4.27

As this instance generates the same results for the original query and the mutant, this mutant is declared as equivalent and this instance is discarded.

The 5 database instances generated to kill non-equivalent mutants are documented in the tables below. First we give the database tuples generated; each test database is one tuple in the `Textbooks` table combined with one tuple in the `Reviews` table. These 5 tuples are given below.

Instance	Textbooks			
	id	**name**	**cost**	**rating**
T1	1	Software Testing	100	4.22
T2	2	Data Mining	NULL	4.23
T3	3	Databases	100	NULL
T4	4	Algorithms	100	4.25
T5	5	Artificial Intelligence	100	4.26

Instance	Reviews		
	id	**user**	**critic**
R1	1	3.45	4.22
R2	2	3.46	4.23
R3	3	3.47	4.24
R4	4	3.48	NULL
R5	6	3.49	4.26

The following table presents the database instances generated to kill the mutants. The last column indicates the number of mutants killed against the mutants executed on that database instance. Note that the mutants killed by the first database instance (generated using the original query) are not executed against other database instances generated and so on.

Textbooks	**Reviews**	**Mutants Killed/Mutants executed**
T1	R1	54/74
T2	R2	6/20
T3	R3	5/14
T4	R4	3/9
T5	R5	4/6

To summarize, 72 non-equivalent mutants of the query in Example 1 are killed by generating the 5 database instances produced by the technique proposed in this paper. The technique identified 2 mutants as equivalent.

In the worst case, in which all mutants are equivalent to the original query, a database instance is generated for each mutant along with an instance for the original query. The time to create 'n+1' database instances is $O(en)$, where 'e' is the execution time of the query and 'n' is the number of mutants. If the database instance of each mutant is executed on all remaining mutants, as done in our evaluation, the worst cast time is $O(en^2)$.

For an additional comparison, the Kaminski approach to mutating the contents of WHERE conditions of SQL queries [16], was used to generate test databases, and these test databases were evaluated against the mutants from the SQL-Mutation tool. Since the WHERE clause in the example is a simple conjunction, the test data generated by the Kaminski technique is equivalent to implementing the MCDC coverage criteria. Of the 74 mutants addressed by our technique, the data from the Kaminski approach killed 66, without identifying the two equivalent mutants. The mutants not killed by the test data are the the 6 non-equivalent mutants generated by NULL mutation operators.

VI. Conclusion and Future Work

Current test data generation techniques for database queries do not guarantee all the following conditions: kill all the non-equivalent mutants; identify equivalent mutants; achieve the previous two conditions for any possible set of mutation operators. This paper presents an algorithm to identify all the equivalent mutants and kill all the non-equivalent mutants generated by any set of mutation operators as long as they belong to the class of conjunctive queries with equalities. A mathematical proof is accompanied to show that this algorithm will distinguish any two non-equivalent conjunctive queries with equalities by generating a counter example and absence of such counter example guarantees equivalence between them. This technique is applied to a simple SQL query and its mutants to demonstrate its functionality. Future research directions for this work include:

- developing similar test data generation algorithms for more useful classes of queries like conjunctive queries with comparisons and aggregate queries.
- exploring ways to minimize the test data required to kill all the generated mutants.
- developing a tool similar to SQL Mutation[6] to automate this test data generation process.
- conducting an extensive study by applying these test data generation algorithms to real life queries.

References

[1] R. Davies, J. Beynon, and B. Jones, "Automating the testing of databases," in *The First International Workshop on Automated Program Analysis, Testing and Verification*. Limerick, Ireland: ACM, June 2000.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[3] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases," in *STOC '77 : Proceedings of the ninth annual ACM symposium on Theory of Computing*. New York, NY, USA: ACM, 1977, pp. 77–90.

[4] A. Klug, "On conjunctive queries containing inequalities," *ACM*, vol. 35, no. 1, pp. 140–160, 1988.

[5] S. Cohen, W. Nutt, and Y. Sagiv, "Deciding equivalences among conjunctive aggregate queries," *ACM*, vol. 54, no. 2, 2007.

[6] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "SQLMutation: a tool to generate mutants of sql database queries," in *Second Workshop on Mutation Analysis (IEEE Mutation 2006)*, Raleigh, NC, November 2006.

[7] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker, "A framework for testing database applications," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000)*. Portland Oregon, USA: ACM SIGSOFT, 2000, pp. 147–157.

[8] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications," in *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China: ACM, 2006, pp. 102–111.

[9] C. Binning, D. Kossmann, and E. Lo, "Reverse query processing," in *ICDE '07: IEEE 23rd International Conference on Data Engineering*. Istanbul, Turkey: IEEE, 2007, pp. 506–515.

[10] M. J. Suarez-Cabal and J. Tuya, "Improvement of test data by measuring SQL statement coverage," in *STEP '03: Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 234–240.

[11] G. M. Kapfhammer and M. L. Soffa, "Database-aware test coverage monitoring," in *Proceedings of the 1st Conference on India Software Engineering (ISEC 2008)*. Hyderabad, India: ACM, 2008, pp. 77–86.

[12] C. de la Riva, M. J. Suarez-Cabal, and J. Tuya, "Constraint-based test database generation for sql queries," in *AST '10: Proceedings of the 5th workshop on Automation of Software Test*. New York, NY, USA: ACM, 2010, pp. 67–74.

[13] T. Y. Chen, M. F. Lau, and Y. T. Yu, "MUMCUT: A fault-based strategy for testing boolean specifications," in *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*. Takamatsu, Japan: IEEE Computer Society Press, 1999, pp. 606–613.

[14] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Information and Software Technololgy*, vol. 49, no. 4, pp. 398–417, 2007.

[15] G. Kaminski and P. Ammann, "Using a fault hierarchy to improve the efficiency of DNF logic mutation testing," in *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009)*, Denver, CO, April 2009, pp. 386–395.

[16] G. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," *Journal of Information and Software Technology*, vol. 53, no. 10, pp. 1137–1152, October 2011.

[17] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *8th International Workshop on Automation of Software Test (AST)*. IEEE, 2013, pp. 111–117.

[18] 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 *IEEE 27th International Conference on Data Engineering (ICDE), 2011*. IEEE, 2011, pp. 1175–1186.

[19] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *International Conference on Software Testing, Verification and Validation (ICST 2013)*. IEEE, March 2013.

[20] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "Efficient mutation analysis of relational database structure using mutant schemata and parallelisation," in *International Workshop on Mutation Analysis (Mutation 2013)*. IEEE, March 2013.

[21] R. Ramakrishnan and J. Gehrket, *Database Management Systems*. New York, NY, USA: McGraw-Hill, Inc., 2002, iSBN:0072465638 9780072465631.

[22] S. Chaudhuri and M. Y. Vardi, "Optimization of real conjunctive queries," in *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ser. PODS '93*. New York, NY, USA: ACM, 1993, pp. 59–70.