Mutation Analysis of Relational Database Schemas

Chris J. Wright

Supervisor: Phil McMinn
External Advisor: Gregory Kapfhammer

A thesis submitted in partial fulfilment of the
requirements for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

September 2015

Abstract

The schema is the key artefact used to describe the structure of a relational database, specifying how data will be stored and the integrity constraints used to ensure it is valid. It is therefore surprising that to date little work has addressed the problem of schema testing, which aims to identify mistakes in the schema early in software development. Failure to do so may lead to critical faults, which may cause data loss or degradation of data quality, remaining undetected until later when they will prove much more costly to fix.

This thesis explores how mutation analysis – a technique commonly used in software testing to evaluate test suite quality – can be applied to evaluate data generated to exercise the integrity constraints of a relational database schema. By injecting faults into the constraints, modelling both faults of omission and commission, this enables the fault-finding capability of test suites generated by different techniques to be compared. This is essential to empirically evaluate further schema testing research, providing a means of assessing the effectiveness of proposed techniques.

To mutate the integrity constraints of a schema, a collection of novel mutation operators are proposed and implementation described. These allow an empirical evaluation of an existing data generation approach, demonstrating the effectiveness of the mutation analysis technique and identifying a configuration that killed 94% of mutants on average. Cost-effective algorithms for automatically removing equivalent mutants and other ineffective mutants are then proposed and evaluated, revealing a third of mutation scores to be mutation adequate and reducing time taken by an average of 7%. Finally, the execution cost problem is confronted, with a range of optimisation strategies being applied that consistently improve efficiency, reducing the time taken by several hours in the best case and as high as 99% on average for one DBMS.

Acknowledgement

I would first like to thank my supervisor, Dr Phil McMinn, who helped to both provide me with the opportunity to undertake this PhD and the which support enabled me to make the achievements described in this thesis. I also owe thanks to my external advisor, Dr Gregory Kapfhammer, for his guidance, enthusiasm and hospitality over the years of my studies. Through their patience, thorough feedback and vast experience I have been able to undertake work that has given me the privilege of presenting at international venues, among other opportunities for which I am incredibly appreciative.

I am also grateful for the personal support of Ryan Bibby, Dominic Rout, Mathew Hall and Sina Shamshiri, who helped to sustain my morale through my course and politely humoured numerous in-depth discussions of my work, as well as helping me take much needed breaks. In addition, I am grateful to the members of the Verification and Testing group for their encouragement and thought-provoking conversations, which provided much needed relief on many occasions. Eternal thanks are also owed to my parents whose support has been essential to me undertaking this PhD. Finally, I wish to thank my wife Laura for her endless understanding, love and patience throughout, including whilst pregnant with our wonderful twin boys, Matthew and Joshua.

Declaration and Publications

This thesis contains original work undertaken at the University of Sheffield between October 2011 and September 2015. Parts of this work have been published elsewhere:

Journal papers:

Phil McMinn, Chris J. Wright, and Gregory M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology*, 25(1):8:1–8:49, 2015

Conference/Workshop papers:

Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2013

Chris J. Wright, Gregory M. Kapfhammer, and Phil McMinn. Efficient mutation analysis of relational database structure using mutant schemata and parallelisation. In *Proceedings of the 8th International Workshop on Mutation Analysis*, 2013

Chris J Wright, Gregory M Kapfhammer, and Phil McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Proceedings of the 14th International Conference on Quality Software*, 2014

Phil McMinn, Gregory M. Kapfhammer, and Chris J. Wright. Virtual mutation analysis of relational database schemas. In *International Workshop on the Automation of Software Test*, 2016. (Accepted)

Additional related work undertaken during this time but outside of the scope of this thesis has also been published:

Cody Kinneer, Gregory M. Kapfhammer, Chris J. Wright, and Phil McMinn. Automatically evaluating the efficiency of search-based test data generation for relational database schemas. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015

Cody Kinneer, Gregory M. Kapfhammer, Chris J. Wright, and Phil McMinn. expOse: Inferring worst-case time complexity by automatic empirical study. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015

Contents

Chapter 1

Introduction

1.1 Overview

While the average person may be unaware of importance of databases, they are a critical piece of infrastructure in modern society, storing data about almost every facet of everyday life – from banking and commerce, to medical records and electronic communication [100]. Information stored in databases has been claimed to be critical tools in political campaigns [21], as well as an important component in mobile devices, Internet browser software and entertainment systems [103]. Recognising the intrinsic value of data has led to calls for organisations to allow open access to the information they store, especially those operating in the public sector. For example, the governments of both the United Kingdom [109] and the United States of America [112] are amongst those who have launched initiatives to release publicly held non-sensitive information, aiming to provide transparency and unlock the potential additional economic value held in this data.

Unfortunately, information and understanding cannot easily be derived from raw data unless it can be reliably stored in a structured way, updated and maintained over time, and retrieved easily. Data that is not stored in a reliable way may become corrupt or incomplete, leading to difficulties in extracting meaningful information. In addition, without structure it may be difficult to discern the meaning of the data, therefore preventing

information being extracted. Likewise, if data is not properly maintained it may lead to misleading information being produced, causing business decisions to be made upon false conclusions – as well as potentially failing to comply with legal obligations. Finally, data must be retained in a format that can be easily retrieved to create useful information, otherwise its inherent value cannot be realised.

A relational database provides a structured means of storing data that meets these requirements, based upon a conceptual model proposed by Codd [27]. Data is organised into columns and tables, which may have references between them, according to a *schema*. This ensures that all stored data must conform to a particular structure and organises data such that it can later be retrieved. Interaction with a database is facilitated through an application known as a *database management system* (DBMS), which allows the creation of a database and the subsequent execution of *queries* against it to retrieve, update and remove items of data. These queries are most commonly expressed in the *Structured Query Language* (SQL), which provides a standardised language for communication with a DBMS.

For data to be considered sufficiently high quality such that meaningful and correct information can be derived from it, thus allowing a business to realise its value, it is essential that it is accurate [115]. Although a schema defines the structure of the data stored in a relational database, defining the tables, column and data types of a database is not enough to ensure that all data inserted into the database is adequately accurate. For example, rows in the database may include empty values – usually referred to as `null` values in databases – or may contain spurious data, like zero or negative values for a `price` column. Such rows may allow for incomplete or inaccurate data to be stored in the database, which may cause faults to manifest in any application that makes use the data in it, such as allowing products to be sold at a loss. Such additional restrictions on what data is acceptable can also be encoded in a schema using a series of *integrity constraints*, which must each be satisfied by a row of data for it to be added into the database. These allow the schema designer to incorporate additional business rules into the database – such as ensuring that a price is always greater than zero, that all individuals can be uniquely identified, or that an aeroplane seat cannot be double-booked. Because integrity constraints are defined at the database level, rather than in each application using the database, they automatically ensure that all applications using the database conform to the same restrictions without the need for each application to reimplement

them, which would be more prone to errors. These constraints are therefore critical to ensuring that the data in a database can be relied upon to be accurate and to not be corrupted by the addition of poor quality data, which may otherwise be allowed by the definition of tables, columns and their data types.

Given the significant impact that integrity constraints may have on the data which can be stored in the database, and therefore in protecting a valuable asset, it is surprising that industry practitioners reportedly do not frequently test the schemas of their databases as recommended [40]. Similarly, while a variety of testing research has focussed on detecting faults in either DBMSs (e.g., [101, 65, 15, 8]) or applications using databases (e.g., [24, 106, 29]), the area of *schema testing* has been left relatively unexplored. Schema testing aims to identify where elements of the schema have been incorrectly specified, such that a database created using it would either allow data to be persisted that is invalid according to some business rules, lowering the accuracy of the stored data as a whole, or preventing valid data from being added to the database, leading to loss of data. These both lead to a decrease in the realisable value of the data stored, which was likely to be costly to obtain and maintain, as a business is unable to rely upon its accuracy or completeness.

To our knowledge, our paper on schema testing [62] was the first to propose a data generation technique specifically for identifying where the integrity constraints of a schema may contain such errors. This applies a meta-heuristic search algorithm to generate data that aims to exercise the constraints within a schema, implemented in a tool called *SchemaAnalyst*. The *coverage* that this automatically generated data achieves is generally determined according to how many constraints were satisfied or violated by at least one row of data. Once the data is executed with a DBMS the tester can examine the rows *accepted* by the schema and the data added to the database, because all constraints were satisfied, or *rejected* by the schema and not added to the database, as one or more constraints were violated. Provided the coverage is high, and therefore a majority of the constraints have been exercised, examining these two sets of data will allow the tester to determine where their schema may have unexpected behaviour and therefore likely contains a fault.

To evaluate the quality of the test data produced by the search-based technique in the SchemaAnalyst tool, we proposed an approach for applying *mutation analysis* to the investigate its fault-finding capability [62]. Mutation analysis is a commonly researched

technique that allows test suites to be evaluated by analysing their ability to distinguish the presence of a series of faults that have been intentionally added to a program (or other software artefact) to create *mutant* programs, each with the aim of subtly altering the program's behaviour. Assuming that the injected faults in these mutants model a suitably wide range of programmer errors, a test suite that can detect a high proportion of these small changes should also detect many genuine errors made by a programmer. Therefore, if the tester is confident the behaviour specified in their test suite is correct, they can be given assurance that such genuine errors are unlikely to be present, and that by extension the program should behave according to its specification. Where a fault is not detected by the test suite, the tester can further refine their test suite by manually producing additional test cases, repeating this process until a satisfactory proportion of mutants can be identified. However, it is possible that one or more of the mutants cannot be detected by any test case – the *equivalent mutant problem* – which may lead the tester to waste time attempting to produce an infeasible test case. Additionally, it is important that the mutation analysis procedure is efficient enough to allow for a potentially large number of mutants to be processed, enabling the analysis of larger programs to be completed in a reasonable time.

This thesis explores a number of important, yet previously unexplored, facets relating to the mutation analysis of integrity constraints contained within relational database schemas, as a means of assessing the fault-finding capability of a series of database rows. This includes confronting the problems commonly associated with mutation analysis – such as the design of a system for generating mutants of database schemas, the automated identification of a number of types of unwanted mutants (including equivalent mutants), and the scalability of analysis in light of potentially large numbers of mutants – in the previously unexplored context of mutating integrity constraints in schemas, proposing and evaluating techniques for each. Domain-specific challenges such as the varying behaviour between DBMSs, due to varying degrees of conformance with the SQL specification, are discussed and their impact managed. In addition, the fault-finding capability of different data generation configurations of SchemaAnalyst are empirically evaluated, to determine how best to produce data to find potential mistakes in the integrity constraints of schemas. Together, these facets yield a complete mutation analysis system for evaluating the capability of a test suite to detect faults in the integrity constraints of a relational database schema, including techniques to reduce both the human and computational cost, as well as demonstrating how this can be used to assess the quality of data produced by an

existing data generation approach.

The remainder of this Chapter first provides a brief background of relational databases and their integrity constraints, as defined using the SQL language. Next, a motivating example for schema testing is described, demonstrating how mistakes in a schema may manifest as faults in applications using a database based upon it. Finally, the novel contributions and structure of this thesis are detailed.

1.2 Relational Database Schemas

Using the relational model of data proposed by Codd [27] a series of values, or *attributes*, relating to one object (e.g., an item of inventory) are stored as *tuples*. A collection of tuples that relate to the same types of object are stored together in a *relation*, where each tuple should usually be able to be uniquely identified. A *query* can then be used to retrieve data from a relation by specifying which attributes are required and any expressions those attributes must satisfy. Values from multiple tuples may be queried together using a *join* operation. Queries can also be used to insert, delete or update tuples, modifying the contents of a relation.

In practical usage, the relational model is commonly applied to define and manipulate databases using a language called *Structured Query Language* (SQL). In SQL, tuples are referred to as *rows*, attributes as *columns* and relations as *tables*. The structure of a database is defined using SQL in an artefact known as a *schema*, which defines the table and columns that will be used to store rows of data. This is expressed as a series of `CREATE TABLE` statements with one statement for each table, containing the definitions for its columns, that are submitted to the database in a query. The SQL language also defines a syntax for the queries used to add rows to the database (`INSERT` statements), retrieve rows matching given criteria (`SELECT` statements with optional `WHERE` clauses), modify values of rows already in the database (`UPDATE` statements), remove rows from the database (`DELETE` statements), and discard the tables of the database (`DROP TABLE` statements). An example of each of these types of query, their effect on the state of a database, or the values that they return when executed for `SELECT` queries, are shown in Figure 1.1.

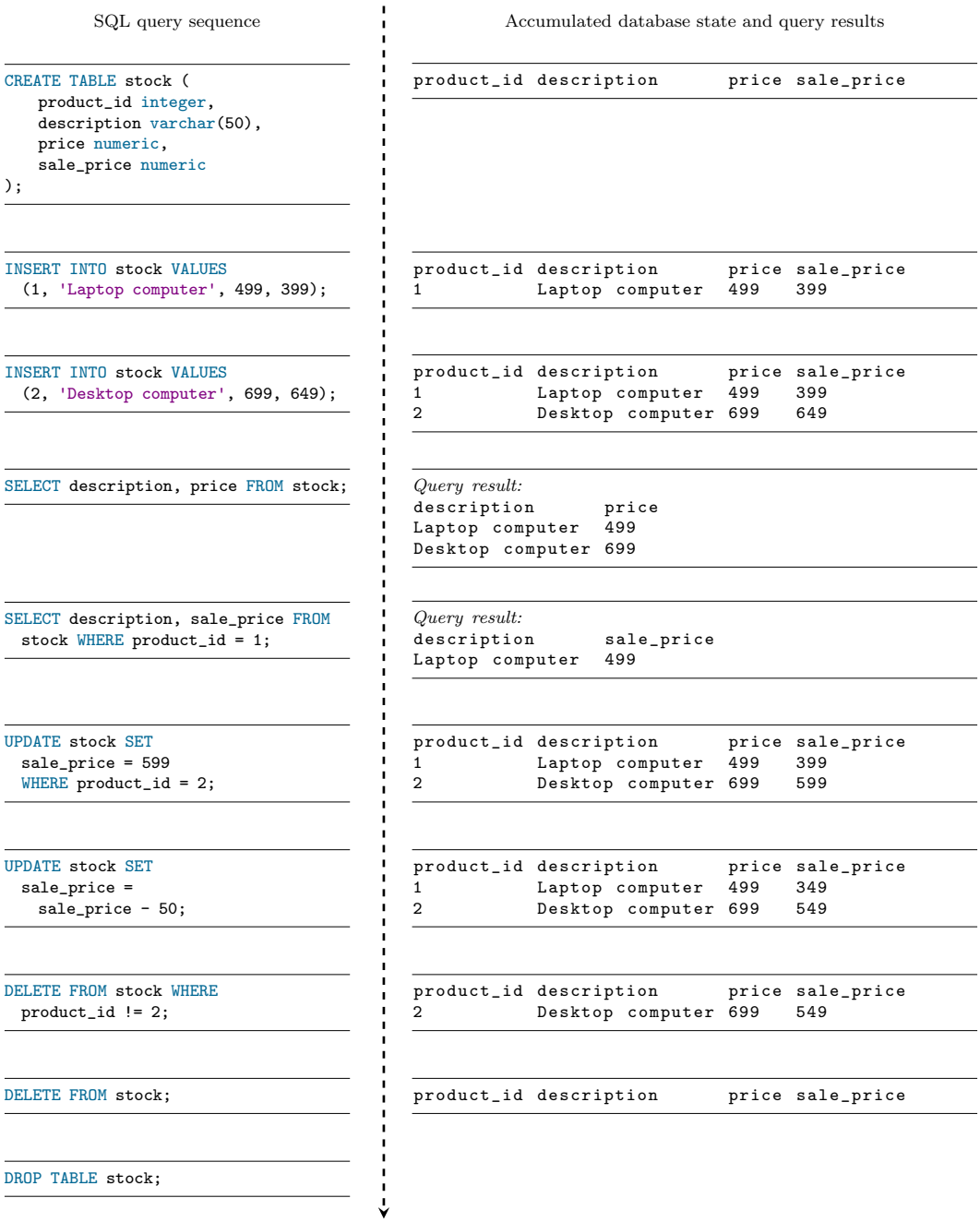


Figure 1.1: Example SQL queries executed in sequence using the SQLite DBMS.

The `CREATE TABLE` query produces an empty database with one table, `stock`, that contains four columns. The `INSERT` queries then each add a row of data to the database. The `SELECT` queries demonstrate how data can be retrieved from the database, including using a `WHERE` clause to limit the rows returned. The `UPDATE` queries show how entries in the database can be updated, including applying changes only to specific rows and referencing existing column values in arithmetic expressions. The `DELETE` queries illustrate the removal of one or more rows of data. Finally, the `DROP TABLE` query shows how a table is discarded entirely.

1.2.1 Database constraints

While the organisation of data into tables and columns provides some structure to the data, it does not allow the schema designer to prevent data being inserted into the database that is invalid according to one or more business rules. Instead, relational databases allow a series of constraints to be defined that must be satisfied whenever data is being added to the database, otherwise the executing `INSERT` statement will fail and the data will not be incorporated into the database.

The simplest type of database constraints are *domain constraints* [100], which limit the value of each column within a row to a specific data type. Specifying the data type of a column is compulsory in each column definition of a `CREATE TABLE` statement, therefore it is not possible to omit a domain constraint – although it is clearly possible to specify the wrong type. The `CREATE TABLE` query in Figure 1.1 demonstrates the definition of domain constraints for each column in the `stock` table. This states that `product_id` may only contain whole-number integer values, `description` must be a character string[1], while `price` and `sale_price` may contain numeric values that include a decimal part.

More complex restrictions on what constitutes valid data for a given database may be specified by including one or more *integrity constraints*, which are described independently of their implementation in a given DBMS by the standards that define the expected behaviour of SQL (e.g., [7]). These are explicitly defined by the schema designer and therefore may experience both faults of commission, where a constraint may overly restrict the rows of data that will be accepted, or faults of omission, where the constraints defined are insufficient to restrict the insertion of data deemed invalid by the business rules. This Section now continues to describe the five types of integrity constraint, providing examples of how each allows the schema design to define what criteria all rows of data must satisfy.

`NOT NULL` constraints

By default, an `INSERT` query may leave any column in an SQL row database unspecified, in which case the special `null` value is used. This signifies that the value is effectively

[1]The type `varchar` is an abbreviation of 'variable length character', a type designed to store strings that may contain a variable number of characters between zero and some maximum number – in this case, the limit is specified as 50 characters, although a default value is used if this is omitted.

unknown, which differs from a value of 0 or the empty string in that the outcome of some boolean expressions may in turn become unknown if any argument is `null`[2]. Alternatively, an `INSERT` statement may include explicitly `null` values, to indicate definitively that a specific column value is not known.

A `NOT NULL` constraint may be defined on a given column to prevent `INSERT` statements either omitting a value or specifying a `null` value for it being accepted into the database. For example, in the `stock` table specified in Figure 1.1, the definition of the column `price` may be amended to '`price numeric NOT NULL`' to ensure that null values are disallowed, and thus meeting the business requirement that all products must have a price specified.

`UNIQUE` constraints

While the relational model of data proposed by Codd [27] specifies that rows in the database should always be uniquely identifiable, the SQL standard does not enforce this requirement. Nonetheless, to allow each row of data to be retrieved, updated and removed without other rows being affected there must be a unique set of attributes that identify only that row. For example, in Figure 1.1 the `product_id` column appears to uniquely identify each row, with it taking the different values of `1` and `2` for the sample `INSERT` statements.

To guarantee that the values taken by one or more columns are unique amongst all other rows in the table, a `UNIQUE` constraint may be specified. This ensures that any `INSERT` statement will be rejected unless the specified column values do not already exist. The SQL language allows this constraint to be defined either when specifying the column name and data type, or elsewhere in the `CREATE TABLE` statement. For example, in the case of the `stock` schema the former would require the column definition to be changed to `product_id integer UNIQUE`, while the latter would involve adding `UNIQUE(product_id)` on a separate line within the `CREATE TABLE` statement.

As mentioned above when describing `NOT NULL` constraints, unless such a constraint is defined for a column, rows may contain the special value of `null`, signifying the actual value is unknown. When evaluating whether one or more columns are unique for

[2]In the case of certain expressions, this requires SQL to operate according to a three-valued logic system where boolean expressions may be either true, false or unknown.

the purposes of a `UNIQUE` constraint, these unknown values are not treated as equal – as the actual value of each occurrence of `null` may in fact be different. Therefore if the `product_id` was defined with a single-column unique constraint as described above, multiple items may be added to the database with an identifying value of `null`.

`PRIMARY KEY` constraints

A `PRIMARY KEY` constraint dictates that one or more columns form a unique identifier for each row and that those columns may not be `null` – essentially combining the restrictions of both a `UNIQUE` constraint over those columns and a `NOT NULL` on each of them[3]. This type of constraint can only be defined once per table and is therefore used for the most common set of columns that uniquely identify a single row, while `UNIQUE` constraints may be defined for other columns that must be unique according to the business requirements.

In the case of the example schema in Figure 1.1 while `product_id` should have unique values for each row in the database, as discussed in the Section immediately prior, it also appears logical that the value `null` should be disallowed – otherwise the database may contain products that it cannot guarantee to be uniquely identified, and so may be sold at an incorrect price if the wrong matching row is returned. In addition, as the `product_id` column appears to be the only column that will always be unique for each product it is a natural choice for a `PRIMARY KEY` constraint, rather than a `UNIQUE` constraint. As with `UNIQUE` constraints, a `PRIMARY KEY` may either be added as part of a column definition (`product_id integer PRIMARY KEY`) or within the surrounding `CREATE TABLE` statement (`PRIMARY KEY(product_id)`).

`FOREIGN KEY` constraints

While separate tables in a schema may be used to avoid duplication of data, it may be necessary to join the rows of different tables together according to one or more matching column values. In some cases, the data in one table can only be understood once such a join has been performed, therefore the rows referenced from one table must always exist in the *referenced* table. For example, expanding upon the schema of Figure 1.1 an `orders` table might be added to record purchases:

[3]As described later in this thesis, although this restriction is specified in the SQL standard [7] the adherence to it differs between DBMSs – for example, the SQLite DBMS does not disallow `null` values for columns within a `PRIMARY KEY` constraint [1].

```
CREATE TABLE purchases (
  order_id integer PRIMARY KEY,
  product_id integer NOT NULL,
  customer_id integer NOT NULL,
  quantity integer NOT NULL
);
```

In this case, each order relates to a purchase made by a specific customer, whose attributes are stored in another table (not shown), to obtain some quantity of a single product, as detailed in the `stock` table. Without a constraint to enforce that both the customer and the product exist, as referenced by the `customer_id` and `product_id` columns respectively, it may be possible to attribute an order to a customer who does not exist or to place an order for a non-existent product.

A `FOREIGN KEY` constraint enforces that the values of one or more columns in a table must correspond to rows which already exist in the referenced table. In the above example, the constraint to limit products to those in the `stock` table would be defined as '`FOREIGN KEY product_id REFERENCES stock(product_id)`' within the `CREATE TABLE` statement for the `purchases` table. This strictly limits the domain of valid values of `product_id` in the `purchases` table to those already added to the `stock` table, such that any `INSERT` queries violating this would be rejected by the DBMS.

`CHECK` constraints

While the prior four types of constraint are specific in their applications, `CHECK` constraints allow a wide range of boolean expressions to be included in a schema. All of these must evaluate to true for any data that is added to the database with an `INSERT` statement or when an `UPDATE` statement tries to modify existing data.

Each `CHECK` constraint defined in a schema may include an arbitrarily complex combination of expressions that may reference the column values of the row an `INSERT` statement is attempting to add to the database. For example, expressions may test whether a value is set to `null`, compare two values – either both column values or a column value and a constant – using a relational operator (i.e., $=$,$\neq$,$<$,$\leq$,$\geq$), test whether a column value is between two limits ('*x* `BETWEEN` *y* `AND` *z*'), or ensure a column takes one value from a given list of values ('*a* `IN (`*b*`,`*c*`)`'). These can then be combined using conjunctions and disjunctions to create more complex logic to reflect detailed business rules.

Continuing the `stock` example table, a number of simple `CHECK` constraints can be specified to ensure that sensible limitations are placed on the data that can be stored. For example, the `stock` table, which has already been refined by the addition of other types of constraints, may be further improved with `CHECK` constraints as follows:

Original

```
CREATE TABLE stock (
    product_id integer PRIMARY KEY,
    description varchar(50),
    price numeric NOT NULL,
    sale_price numeric NOT NULL
);
```

With `CHECK` constraints

```
CREATE TABLE stock (
    product_id integer PRIMARY KEY,
    description varchar(50),
    price numeric NOT NULL,
    sale_price numeric NOT NULL,
    CHECK ((price > 0) AND
      (sale_price > 0)),
    CHECK (sale_price <= price)
);
```

These additional constraints enforce two sensible business rules. Firstly, both the price and sale price of an item must be greater than zero, ensuring a negative or zero value cannot be specified by mistake. Secondly, the sale price of an item must always be less than or equal to the normal price, ensuring that whenever the customer is offered the sale price they cannot be charged more than usual.

1.2.2 Summary

This Section has provided background of how SQL can be used to define the structure of a relational database, in terms of table, columns and their data types, and integrity constraints. The five types of constraints that are commonly found in SQL schemas have been discussed, with descriptions of the types of data they help prevent from being added to the database and a worked example showing how these can be applied in a simple scenario. Next, a motivation for schema testing is provided to further explain the importance of detecting and resolving mistakes made when defining the integrity constraints of the schema.

1.3 Motivation for Schema Testing

As previously mentioned in Section 1.1, there has been little research into techniques for schema testing, meaning it is difficult to determine where faults may have been included in the integrity constraints of a relational database schema. When compared to the efforts in developing approaches to test both DBMSs themselves (Section 2.2.1) and the database applications that use them (Section 2.2.2), this leaves a conspicuous opportunity for errors to occur because of mistakes made early in the development of such applications, when a data model is transformed into a database schema and constraints are defined. Given that these constraints protect against the database becoming corrupted, leading to possible loss of data or an increased difficulty in extracting meaningful information – and therefore generate value – from the database, the correct definition of these constraints is critical to the success of an application. In addition, if constraints overly limit the data that can be stored, important data may be lost that should have been persisted into the database, indicating that both faults of *omission* and *commission* [4] should be considered. Moreover, while database application testing techniques may be able to detect these problems in some circumstances and resolve them in the application layer of the application, rather than the data layer, this relies upon all applications sharing a given database to follow the same technique and apply the same resolutions – which is, at best, a potentially unreliable and expensive prospect.

Schema testing aims to aid in detecting faults located in the definition of the structure of the database and its constraints. This thesis focusses on how mutation analysis can be used to evaluate schema testing techniques that exercise the constraints in a schema, through execution of sequences of `INSERT` statements, and how to ensure this is performed both efficiently and effectively. This Section now continues with a description of a schema, possible faults that may easily be made when defining the integrity constraints and their impact on an application using a database produced with this schema, and how schema testing would help detect the faults before they manifest as potentially hard to detect bugs in the application.

Correct schema

```
CREATE TABLE stock (
    product_id integer PRIMARY KEY,
    description varchar(50) NOT NULL,
    price numeric NOT NULL,
    sale_price numeric NOT NULL,
    CHECK ((price > 0) AND
        (sale_price > 0)),
    CHECK (sale_price <= price)
);

CREATE TABLE customer (
    customer_id integer PRIMARY KEY,
    name varchar(100) NOT NULL,
    address varchar(100) NOT NULL,
);

CREATE TABLE purchases (
    order_id integer PRIMARY KEY,
    product_id integer NOT NULL,
    customer_id integer NOT NULL,
    quantity integer NOT NULL,
    FOREIGN KEY product_id
        REFERENCES stock(product_id),
    FOREIGN KEY customer_id
        REFERENCES customer(
            customer_id),
    CHECK (quantity > 0);
);
```

Possible faults

```
CREATE TABLE stock (
    product_id integer PRIMARY KEY,
    description varchar(50) NOT NULL,
    price numeric ~~NOT NULL~~,
    sale_price numeric NOT NULL,
    CHECK ((price > 0) AND
        (sale_price > 0)),
    CHECK (sale_price <= price)
);

CREATE TABLE customer (
    customer_id integer UNIQUE,
    name varchar(100) NOT NULL,
    address varchar(100) NOT NULL,
);

CREATE TABLE purchases (
    order_id integer PRIMARY KEY,
    product_id integer NOT NULL,
    customer_id integer NOT NULL,
    quantity integer ~~NOT NULL~~,
    FOREIGN KEY product_id
        REFERENCES stock(product_id),
    FOREIGN KEY customer_id
        REFERENCES customer(
            customer_id),
    CHECK (quantity >= 0)
);
```

Figure 1.2: An example of an SQL schema and several faults in integrity constraint definitions that would be identified through schema testing.

1.3.1 Motivating example

The schema in Figure 1.2 provides a more complete listing of the examples used thus far to describe the various constraints in SQL. This produces a database that can be used to track items that are offered to be sold, customers who may buy them and record purchases that have been made. While it contains some assumptions for the sake of clarity, such as each order only permitting one type of product from the `stock` table and customers being limited to a single `address`, it still provides ample scope for simple mistakes in constraints to have a significant effect. This is because the constraints in the schema define a series of business rules that ensure any application using this database functions correctly. A series of four possible programmer mistakes highlighted on the right-hand side of Figure 1.2 are now described, in terms of both their impact and how schema testing would enable them to be detected.

Highlighted on lines 4 and 21, the omission of `NOT NULL` constraints can allow `null` values to be inserted into the database in place of actual values. In the first case (line 4), an application using the database may fail to specify the price for an item – for example, if a user does not enter a value in the relevant field when adding a new item of stock. This `INSERT` statement would still satisfy the remaining constraints of the schema and a `null` value would be added to the database in the `price` column. This value may result in faulty behaviour in the application logic that allows a customer to purchase an item for no charge, unless this logic is thoroughly tested with `null` values being returned by the database. In the second case (line 21), a `null` value would be allowed to be provided as the quantity in an order. Again, this may cause a fault in the purchasing portion of the application, where the number of items to include in an order may become undefined, leading to unpredictable application behaviour. In both cases, thorough testing of the integrity constraints contained within the schema should detect such omissions, by generating data to test both `null` and non-`null` values for each column[4]. Upon reviewing this data, a tester would be able to identify that both the `price` and `quantity` columns should not be able to accept `null` values, thus determining both faults in the schema.

The fault on line 12 represents the case where a `UNIQUE` constraint has been used in place of a `PRIMARY KEY` constraint. If the programmer is familiar with a DBMS that does not infer an implicit `NOT NULL` constraint on the columns of a `PRIMARY KEY` (e.g., SQLite), but is using an alternative DBMS that does so according to the SQL standard (e.g., PostgreSQL), then they may mistakenly believe these constraints are equivalent. However, by permitting `null` values to be specified for the `customer_id` column it may become impossible to retrieve the details of a customer where this has been mistakenly omitted when adding them to the database, if the application using the database relies upon this key value to identify each row. As with the `NOT NULL` constraints, schema testing via generating `null` and non-`null` values would expose this fault to the tester, who would be able to identify from the generated data that `null` values for the `customer_id` column are being allowed into the database.

Finally, the fault on line 26 shows a simple mistake where the wrong relational operator has been specified, with $\geq$ being used instead of $>$. Similarly, this mistake may

[4]A means of generating data according to this strategy as implemented in the SchemaAnalyst tool [76] is described later in Section 3.5.1 and its effectiveness evaluated as a novel contribution of this thesis in Chapter 5.

cause errors to occur in an application where it may be assumed that all purchases must correctly refer to an item from the `stock` table, but does not check the quantity is correctly set and allows a value of zero (which still satisfies the `NOT NULL` constraint even if this is not omitted). Testing a `CHECK` constraint requires the generation of data to both satisfy and violate the constraint. In this case, selection of boundary values that 'just' satisfy or violate the expression would help the tester determine the erroneous case – for example, the tester would recognise that -1 should not be accepted and is correctly rejected by the schema, and that 0 should not be accepted but is incorrectly accepted by the faulty version of the schema. This would help them determine that the `CHECK` constraint is poorly defined and requires modification, leading them to select the correct operator.

1.3.2 Application of Mutation Analysis

While a tester may conceive a strategy for producing test cases that exercise those parts of the schema thought to be most important, without a more rigorous approach they may be liable to forget to include tests for all constraints in the schema. However, this greatly increases the likelihood of small faults remaining undetected until later in the development of the program using a database produced from the schema, likely increasing the cost of remediation significantly. Mutation analysis provides a more guided process for developing a test suite that has a high likelihood of identifying faults in the program, based upon measuring a test suite's ability to detect a set of mutants with known faults. Where a test suite is able to recognise all such mutants, the tester can be assured that their test suite has a high fault-finding capability, assuming the faults applied to produce the mutants are representative of real programmer faults.

Proposed technique

To evaluate the quality of test suites produced for testing the integrity constraints of relational database schemas, such that different data generation techniques could be compared, I proposed a mutation analysis approach for relational database schemas [62]. By executing each of these `INSERT` statements with the original, unmodified schema and then each of its mutants, produced by modifying its integrity constraints, and comparing

which `INSERT` statements are accepted or rejected, it is possible to determine whether the fault of each mutant is detected by the test suite or not. This allows a mutation score – the proportion of mutants detected out of all mutants produced – to be calculated, where a higher score suggests a higher fault-finding capability. The basic steps of this mutation technique, specified more formally in Section 7.2, are as follows:

1. Produce mutant versions of the schema by applying mutation operators.
2. Create a database with the original schema by executing `CREATE` statements.
3. Execute each `INSERT` statement in the test suite against the database.
4. Remove the database from the DBMS with `DROP` statements.
5. For each mutated copy of the schema:
   (i) Create a database with a mutant schema by executing `CREATE` statements.
   (ii) Execute each `INSERT` statement in the test suite against the database.
   (iii) Compare the acceptance of each `INSERT` statement to the acceptance when using the non-mutated schema. If there are any differences the mutant is killed[5]).
   (iv) Remove the database from the DBMS with `DROP` statements.
6. Calculate the mutation score as the proportion of mutants detected.

This technique – hereinafter referred to as the *Original* technique – is pivotal to each of the contributions of this thesis. It generates mutants of relational database schemas according to the operators in Chapter 4, which allow a mutation analysis experiment to be performed to measure the fault-finding capability of different schema coverage criteria to be evaluated in Chapter 5. However, as with the application of mutation analysis to other domains, it is possible that some of the produced mutants reduce its efficiency and effectiveness, which in turn decreases the usefulness of the technique in a practical setting. Therefore, in Chapter 6, I describe a series of algorithms to automatically detect

[5]In the context of schema constraint mutation, a mutant is described as *killed* if at least one of a series of `INSERT` statements accepted by the original, non-mutant schema is rejected by the mutant, or vice-versa.

many of these, and empirically evaluate the effectiveness of an implementation of them. Due to the potentially large number of mutants produced as the size of the mutated artefact increases, mutation analysis can become prohibitively expensive, requiring either significant computational resources and long execution times, or both. This problem is equally present for mutation analysis of relational database schemas as it is for programs. Therefore, in Chapter 7 I discuss how the Original technique can be made more efficient by implementing a series of different time-saving approaches, to ensure mutation analysis can be scaled to large schemas in a reasonable time, and compare their impact in an empirical study.

1.3.3 Summary

This Section has provided an overview of how small mistakes in even relatively simple schemas can cause faults to occur in applications that use them to produce databases. As the scale of the schema increases, the scope for such mistakes is likely to increase, their impact may worsen, and detecting them through database application testing could become more expensive. For each mistake, a means of detecting it through schema testing is described, motivating further investigation of schema testing techniques and evaluation of their effectiveness. To evaluate test suites produced for schema testing, a mutation analysis approach for relational database schemas was discussed, which is central to the contributions of this thesis. The contribution Chapters of this thesis explore numerous facets relating to this approach, exploring both its efficiency and effectiveness, including tackling a number of challenges usually faced when applying mutation analysis, proposing and evaluating a number of solutions to them.

1.4 Contributions of this Thesis

The novel work of this thesis, produced according to the above motivations, can be divided into the following contributions:

C1: Schema mutation operators – I formally describe a collection of mutation operators, implemented as part of a mutation framework, designed to model potential programmer mistakes when defining integrity constraints in relational database

schemas, including operators leveraging static analysis to produce only valid `FOREIGN KEY` mutants, as well as definitions of the predicted productivity of each operator. In each case, worked examples are also given to demonstrate how the operator behaves for a sample schema (Chapter 4).

C2: Empirical evaluation of coverage criteria for database schemas – An empirical investigation determining how the choice of coverage criteria influences the effectiveness of generating data for testing relational database schemas, including an analysis of the types of mutants specific criteria are adept at killing and how combinations of criteria can improve test suite effectiveness (Chapter 5).

C3: Algorithms for removing ineffective mutants and an evaluation of their impact – Definitions of three types of undesirable mutants produced when mutating relational database schema, a series of patterns and detection functions for automatically identifying them, and the results of an empirical study to determine the impact this has on the accuracy and efficiency of mutation analysis (Chapter 6).

C4: Techniques to improve the efficiency of mutation analysis for schema testing – The specification of a number of novel mutation analysis techniques, implementing domain-appropriate equivalents to techniques previously used for program mutants, as well as optimisations specific to the domain of relational databases (Chapter 7).

C5: An evaluation of mutation analysis techniques for schema testing – An empirical evaluation of several optimised mutation analysis techniques to determine their efficiency when applied to a wide variety of schemas and numerous DBMSs as well as their accuracy with respect to the standard, unoptimised technique (Chapter 7).

1.5 Thesis Structure

This thesis relates each of the above contributions through the following structure, the links between which are highlighted in Figure 1.3:

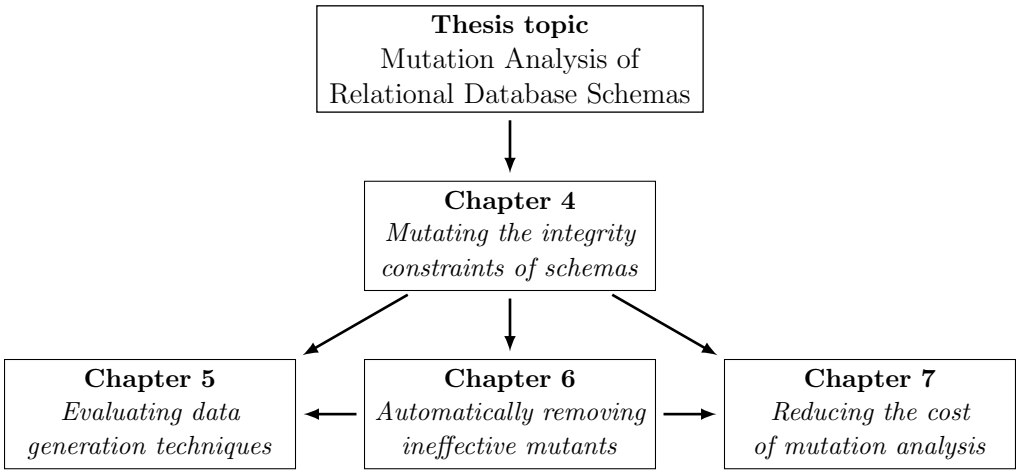


Figure 1.3: An overview of the connections between the Chapters of this thesis.

Chapter 2: "Literature Review" explores literature relating to both database testing – including DBMS testing and database application testing techniques – and mutation analysis – covering the theory underpinning mutation, known challenges such as the equivalent mutant problem and approaches to tackle them, and techniques attempting to improve the efficiency of mutation analysis to enable its use in practical settings.

Chapter 3: "The SchemaAnalyst tool" describes the implementation of a schema testing technique as a tool which is used throughout the Chapters of this thesis. The mutation framework contributed as part of this thesis is described, which is used later in Chapter 4, as well as the test data generation system that is evaluated in Chapter 5.

Chapter 4: "Mutation Operators for Relational Database Schemas" details a collection of novel mutation operators that can be used to inject faults into the integrity constraints of a relational database schema to produce the mutants required by the mutation analysis approach described in Section 1.3.2. These model a wide range of faults in the constraints of an SQL schema, including `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE` constraint, `NOT NULL` constraint and `CHECK` constraint mutants. Where applicable, static analysis approaches are used to prevent the production of semantically invalid mutants. For each operator, a formal algorithm is described in terms of manipulation of the SchemaAnalyst intermediate representation of SQL (Section 3.3), as well as a definition of the operator's productivity.

Chapter 5: "Evaluating Coverage Criteria for Relational Database Schemas Using Mutation Analysis" makes use of these mutation operators and the proposed mutation analysis technique to evaluate the fault-finding capability of test suites produced by difference coverage criteria, which are each designed to exercise the constraints of a relational database schema in different ways. By examining which seeded faults the produced test suites are able to detect, the coverage criteria can be compared in terms of how likely they are to identify genuine errors made when creating a relational database schema. Through analysis of the types of mutants killed most readily by certain operators, a series of combinations of coverage criteria are also evaluated, to determine which produce the overall most effective test suite of SQL `INSERT` statements.

Chapter 6: "Automatically Identifying Ineffective Mutants" more closely examines the types of mutants produced by the mutation operators defined in Chapter 4, highlighting three classes of mutant that distort the results of mutation analysis – equivalent, redundant and quasi-mutants – and details algorithms that can automatically identify and remove these, prior to analysis. Implementations of these algorithms are then evaluated in an empirical study to quantify the effect this removal has on the efficiency of mutation analysis, as well as the accuracy of the mutation score obtained.

Chapter 7: "Improving the Efficiency of Mutation Analysis for Relational Database Schemas" proposes how optimisations can be used to vastly improve the efficiency of mutation analysis of relational database schemas to evaluate schema testing techniques, ensuring it is practical for both large scale empirical experiments such as that in Chapter 5 and practical application. These optimisations are generally inspired by those applied to mutation analysis of programs and other software artefacts, although implement strategies specifically tailored to the domain of schema testing. These are then compared and evaluated through an empirical experiment, to determine which optimisations are able to maintain effectiveness – producing the same results as the unoptimised Original technique – while improving efficiency by minimising the cost of execution.

Chapter 8: "Conclusions and Future Work" contains final remarks about the work in this thesis, summarising the contributions of the prior Chapters and providing possible directions for future research in the area.

Chapter 2

Literature Review

2.1 Software Testing

Given that software is generally written by humans, it is reasonable to expect that as the number of lines of code increases, occasional mistakes become inevitable. While these mistakes may have relatively harmless consequences – for example, causing minor annoyance to a user who must spend a few moments to restart a crashed application – in some cases their impact may be catastrophic – such as the explosion of an Ariane 5 space rocket, worth approximately $500 million, only 40 seconds after launch [49]. By testing software, it is possible to gain confidence that such mistakes can be identified and fixed prior to deployment of the software, usually during or shortly after development, reducing the risk of costly ramifications. As a result, software testing is an essential part of software development, taking up to half of the overall development time according to some estimates [80].

This Section now continues by first describing a number of important terms and concepts commonly used in the context of software testing, including throughout the remainder of this thesis. The different types of testing technique are also discussed, followed by a discussion of metrics that can be used by testers to guide the development of effective tests. Finally, a brief description of mutation analysis, its relation to test development and software testing experiments is given.

2.1.1 Basic Concepts

A number of important terms used throughout software testing literature are described below. Although there are slight variations between definitions given by different software testing researchers, the following definitions provide the generally accepted meanings according to a number of prominent authors in the field (e.g., Ammann and Offutt [4], Burnstein [20], and Myers and Sandler [80]).

Where a mistake is made in the source code of a program this is referred to as an *error*. These may have various causes, such as the programmer misunderstanding the program's requirements, misusing a particular programming construct or simply making a typographical mistake. If the presence of an error causes a program to behave abnormally, such that it no longer meets its specification, this is known as a *fault* – a term used interchangeably with *bug* or *defect*. Not all errors will manifest as faults – for example because they are located in code that is not executed, or do not impact upon the externally observable behaviour of the program. Finally, if the fault is executed during the running of the program causing it to behave incorrectly, it is exposed externally as a *failure*.

Software testing aims to detect faults through the execution of *test cases*. These encode the intended behaviour of the program, specifying that executing part or all of the program should produce some *expected* result, which may be expressed as an *assertion*. For each of these assertions, the expected result is compared with the *actual* result obtained by running the program, with any violated assertion indicating the presence of a fault – or, an incorrectly specified test case. In the former case, the tester should investigate further to determine the underlying error(s), while in the latter they should correct the mistake in their test. The collection of test cases for a program are usually combined into a *test suite*, ideally including tests executing a wide range of its functionality. Where some piece of functionality or section of code is executed by at least one test, it is said to be *covered* by the test suite.

When selecting a testing technique, it is not necessarily assumed that the tester will have the specialist knowledge to understand the internal implementation of the program. This reduces their need for in-depth training and a thorough understanding of the algorithms applied in the code of the program. In *black-box* testing techniques, the tester is left unaware as to the inner workings of the program, comparing only the outputs it

produces when given various inputs. For example, the tester may use values derived from the formal specification of the program, if one exists. Unfortunately, this can limit the tester's ability to give assurance that no faults are present, unless they exhaustively test the program for all inputs [80] – usually a prohibitively expensive task for even small programs. To avoid this, *white-box* testing techniques assume the tester is able to inspect the implementation of the program and use this to inform their choice of test inputs. For example, they may observe literal values used in conditional statements or loop termination conditions and select values that test for 'out by one' errors (where the correct literal value is ± 1 from the stated value). They may also use a systematic approach to ensure that all parts of the program are executed, according to some notion of *test coverage* – what parts of the program have been tested by a given test suite (see Section 2.1.2 for a more detailed description). Unfortunately, compared to black-box testing, this generally requires the tester to gain a more detailed understanding of the program's implementation, which may present a significant human-time cost. By balancing the positive and negative elements of black-box and white-box testing, *grey-box* techniques allow the tester to use partial information about the implementation of the program and combine this with higher-level specifications of the program's expected behaviour to produce tests.

2.1.2 Strategies for Software Testing

Coverage criteria

To ensure that the developed test suite covers a sufficiently broad proportion of a program's functionality, testers may base their testing strategies upon *coverage criteria*. These provide a quantifiable way of determining how well a test suite exercises the program, and therefore in theory the likelihood of it detecting any faults present. Applying these, a tester can identify a target *coverage level* they wish to achieve, where a higher target will involve a greater effort in developing the test suite, but also increase the likelihood that faults are detected.

When using white-box testing techniques, where the program code is available to the tester, a series of *structural coverage* criteria [4] can be employed, which measure what parts of the program tests have executed, to estimate the quality of the test suite. The most basic of these criteria, *statement coverage*, requires that each statement of the

program is executed at least once. However, it is not difficult to devise a simple example where this does not execute all behaviour of a program. Consider the following simple example function, which calculates the new price for an item by applying a fixed discount, but ensures the minimum value of new price is 10:

```
int reducePrice(int price, int discount) {
  int newPrice = price - discount;
  if (newPrice < 10) {
    newPrice = 10
  }
  return newPrice;
}
```

In this case, statement coverage can be satisfied by any input `x` where '`x` < 10'. However, this fails to test the function's behaviour when the '`if`' condition is not satisfied, meaning the implementation may contain a fault in that case. While in such a simple example it is clear by inspection that there are no such errors, this weakness of statement coverage persists in more complicated programs that contain any similar conditional logic. To tackle this, *decision coverage* (DC) (also known as *branch coverage*) dictates that test cases must be provided to satisfy and negate the boolean expression of each conditional branch of the program. Therefore, in the above case, two tests would be required – one where '`price` $-$ `discount` < 10', and another where '`price` $-$ `discount` ≥ 10'. Expanding upon this for boolean expressions that are composed of multiple sub-expressions, *condition coverage* states that a test suite should include tests where each of these sub-expressions evaluates to true and false. For example, with the expression '`(a < 5 && b > 10)`', the test cases required would be '`(a < 5, b > 10) = (true, true)`', '`(a < 5, b < 10) = (true, false)`' and '`(a > 5, b = ?) = (false, not evaluated)`'. In the latter case, it is assumed that the programming language used implements *short-circuiting* of logical operators, where the right hand-side of an '`&&`' (logical AND) expression will not be evaluated if the first operand evaluates to '`false`'. Therefore, the choice of value for `b` is not restricted in this case. Combining decision coverage and condition coverage, *condition/decision coverage* (C/DC) ensures that both each overall top-level boolean expression must be evaluated to true and false, as well as that each sub-expression has also taken both true and false values. For the purposes of testing safety-critical applications, *modified condition/decision coverage* (MC/DC) [25] refines this definition to additionally

specify that tests must ensure each sub-expression independently affects the overall result (i.e., causes it to evaluate to both true and false). While these coverage criteria cannot be used directly for the testing of relational database schemas, Section 3.5.1 briefly describes a series of criteria specifically designed for this purpose by McMinn et al. [76]. The fault-finding capability of each of these criteria is evaluated as a contribution of this thesis in Chapter 5.

Mutation analysis

Another means of evaluating the quality of a test suite is *mutation analysis*, which measures how effective it is at identifying a series of automatically inserted faults. Assuming that these faults are representative of those caused by programmer errors, a test suite that can identify a high proportion of the added faults is also likely to identify many real faults, if any are present. Due to its highly automated nature and ability to model a wide range of faults, mutation analysis has been used in many software testing experiments (e.g., [2, 32, 50, 62, 83, 106]) and has been shown to be an effective means of measuring the fault-finding capability of a test suite for real programs [5, 6, 60]. Mutation analysis is used to compare coverage criteria for testing the integrity constraints of relational database schemas in Chapter 5. A more thorough description of mutation analysis and discussion of relevant literature follows in Section 2.3.

2.1.3 Summary

This Section has outlined a number of basic concepts underlying software testing, including definitions for a range of common terms used in the literature and throughout this thesis. Next, a series of logic coverage criteria for testing programs were described, which help guide the tester to develop tests that exercise all parts of a program, according to different test requirements. Using similar concepts, a set of coverage criteria specifically designed to test the integrity constraints of a relational database schema [76] are described later in Section 3.5.1. Finally, mutation analysis was briefly introduced as a means of evaluating the quality of a test suite, such that suites produced by different coverage criteria can be compared. This is described in greater detail in Section 2.3, and is applied to compare the different integrity constraint testing coverage criteria in Chapter 5. The

following Section describes the more specific problem of database testing, which discusses techniques developed to test both the database management system used to host with databases, and programs that interact with those databases.

2.2 Database Testing

Many real-world applications involve the storage of data such that it can be retrieved, modified and removed easily, while ensuring any loss of data is minimised. This commonly involves a database management system (DBMS) to create a relational database, which stores the data in a structured way according to a schema. The schema is usually expressed in the SQL language and specifies how the data is organised into tables and columns, what types of data can be stored in each column and what constraints any data being stored must satisfy. These constraints may specify that row values must uniquely identify a single row in a table (`PRIMARY KEY` and `UNIQUE` constraints), that they cannot omit specific columns (`NOT NULL` constraints), that they must validly refer to values in another table (`FOREIGN KEY` constraints), or that they satisfy some arbitrarily complex boolean expression (`CHECK` constraints). By applying these constraints the DBMS can ensure that the data stored in the database is of a sufficient quality, preventing incomplete or erroneous data being allowed to be stored in the database. Once stored in the database, data is retrieved, modified and removed by submitting queries to the DBMS, which first checks that the query is valid and then applies it to the database, either returning those rows that were requested or changing the state of the database.

Given the importance of data in many applications, it is important to test that the database component performs as expected. Literature in this area can be divided into two different testing activities – DBMS testing and database interaction testing – both of which are described in this Section. DBMS testing (Section 2.2.1) aims to ensure that the DBMS returns the expected results for given queries. Database interaction testing (Section 2.2.2) investigates whether an application that stores data using a database works correctly.

2.2.1 DBMS testing

If a DBMS exhibits faulty behaviour, then any application that relies upon it to store data in a database would in turn be prone to unreliable behaviour, leading to possible data loss or inconsistency – both of which may prove costly if undetected. It is therefore unsurprising that DBMS testing has been focussed upon by numerous researchers (e.g., [11, 65, 94, 101, 104]). This testing activity requires three artefacts – a number of rows of data to insert into the database, a set of queries to execute against that data, and a series of expected results for those queries. The remainder of this Section first discusses a number of techniques for generating database queries, then describes a variety of proposed approaches for generating sample rows of data, both for the purposes of DBMS testing. In each case, where applicable, the process applied for determining the expected results is also detailed.

Query generation

To test whether a DBMS correctly returns the expected rows when given a query to execute against a database – which is usually the most common operation performed on a database – it is necessary to produce one or more queries. Due to the potentially huge number of queries possible for a given database schema, it is important to both test a wide range of different queries – by selecting a useful range of queries according to some criterion – as well as producing queries by more systematic means – for example enumerating all possible queries for a given schema or targeting certain error prone parts of the DBMS. The former cases are categorised below as 'sampling' approaches [101, 11, 94], which try to produce a wide variety of queries, while the latter as referred to as 'systematic' approaches [65]. Combining techniques from both of these categories may help ensure that a wide range of possible faults in a DBMS can be identified, to give assurance that its behaviour is correct.

Sampling approaches The *RAGS* (Random Generation of SQL) system, presented by Slutz [101], enables a tester to quickly produce a large number – potentially thousands – of SQL `SELECT` queries according to a provided schema via an automated process. This aimed to improve coverage of the massive input domain of possible SQL statements when

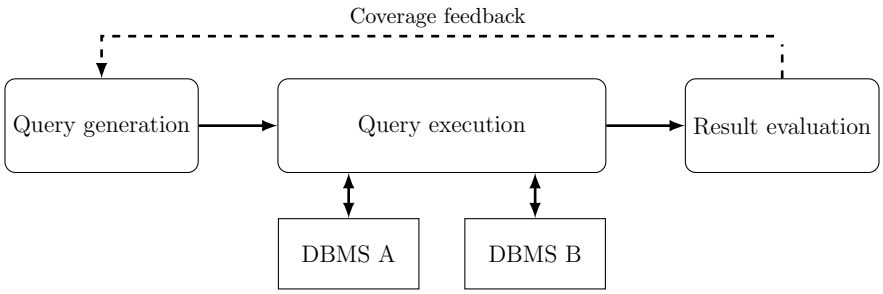


Figure 2.1: A generalisation of the technique used by Slutz [101] to generate `SELECT` queries, and the addition of coverage feedback contributed by Bati et al. [11].

compared to the use of manually created test SQL statements alone. To achieve this, it traverses the parse tree of a given SQL statement and randomly introduces additional elements, to produce a set of more complex `SELECT` statements. This may include adding a wide range of valid SQL constructs such as modifying column lists, table lists, subqueries, expressions in `WHERE` clauses, and grouping clauses. Which of these elements are added is based upon values produced randomly, according to a random seed to allow reproducibility, until a configurable depth is reached. The generated queries can then be used with a database pre-populated with data (e.g., data produced during other testing activities or sampled from a production system), with results being compared between different DBMSs or varying versions of the same DBMS. For example, comparisons were made between the results obtained by systems produced by multiple different software vendors, allowing the results they return to be compared. If each of the systems under test return the same result then either none of the systems contain an error in the processing of the query or, less likely, they all do. The RAGS tool also allows results of these queries to be stored for different versions of a DBMS to be persisted, to allow for regression testing to be performed. Applying this tool to produce 2,000 `SELECT` statements for four different DBMSs indentified a possible genuine programmer error in a commercially released system, demonstrating the industrial applicability of the approach. However, given the random nature of the RAGS tool it is not possible to sufficiently guide the generation of statements to explore a specific part of the overall query space. Additionally, given the requirement for comparing results between multiple systems, the tool is only able to produce queries that are valid in all DBMSs under test. Nonetheless, the RAGS tool is able to quickly produce many valid SQL statements – up to three million per hour – that have been shown to help reveal potential bugs in a DBMS, reducing the risk of faults in

applications using a database.

Improving upon the RAGS system, Bati et al. [11] proposed the use of a genetic algorithm to guide the `SELECT` query generation process to meet particular coverage goals. This functions by producing a 'pool' of queries that are each ranked according to whether they are likely to meet these goals, then proceeding to modify those ranked most highly according to a number of operations – for example, mutating the contents as with the random generation in the RAGS tool, or combining multiple of these statements through joins, subqueries and unions. The fitness function that guides the generation of these queries encourages the production of `SELECT` statements that lead to the exploration of unusual execution paths, and discourages the production of cases that always return empty data sets. Coverage is measured by recording the code paths that have been executed for each query, by augmenting the DBMS with additional logging options to output which paths have been executed. Statements that improve upon those used to create it according to the fitness measurement, statement length or running time, replace them in the pool of queries. This process is then repeated until the pool stabilises for the given fitness function. By varying the fitness function used, it is then possible to focus the areas of code executed by the queries, thus allowing testing to be focussed on more specific parts of the DBMS – for example, those which have been under active development since previous testing, and are therefore much more likely to contain bugs. As with the RAGS system, this genetic algorithm (GA) based system also requires another DBMS to compare the results produced by the system-under-test with, thus avoiding the need to determine the correctness of returned results by other means. The evaluation of this approach involved comparing its ability to produce `SELECT` statements that achieved good coverage of the DBMS to those generated by a random technique. This revealed that the GA-based system was able to achieve the same coverage of DBMS functions within significantly less time than the random approach, and achieved $\sim$29% higher coverage than the random approach when given the same time. This demonstrates its ability to test a DBMS more thoroughly given the same resources, as well as allowing the testing to be focussed more accurately if specific components have a higher likelihood of containing faults.

A similar tool, *QGEN*, created by Poess and Stephens [94] is able to generate queries by the user expressing them using a template language that can then be sampled from to provide instantiated versions of queries, which can be automated by combining it with

```
1 -- Declare substitutions
2 DEFINE
3   year = RANDOM (1950, 2000, UNIFORM);
4
5 -- SQL statements
6 SELECT *
7 FROM person
8 WHERE person.birthyear = year;
```

Figure 2.2: An example query template using the approach of Poess and Stephens [94]. In this case, the variable `year` is configured to take random values between 1950 and 2000, according to a uniform distribution. This is substituted into the `WHERE` clause of the query causing each query to select all rows from `person` with a the given birth year.

a data generator [104]. Using a query template containing variables that are randomly instantiated, a predictably distributed set of queries can be generated, which can then be used to compare the performance of different database systems. This enables the detection of particularly important performance gains or losses between varying versions of a DBMS, which may help identify possible efficiency regressions prior to a new version being used in a production environment. The variables in a template can be instantiated with either uniform or normally distributed values, and may also be nested within each other to produce more complex queries. An example of a simple query template is shown in Figure 2.2. While it was shown by Poess and Stephens that the template approach could accurately model different types of realistic distributions – for example, how sales may be more heavily weighted to certain times of year – it hasn't been thoroughly evaluated in an empirical evaluation. However, anecdotal evidence suggested that the technique is efficient, and offers a means of producing data that can target different parts of the DBMS more effectively than random generation – provided that the tester is able to construct templates with a suitable set of substitutions.

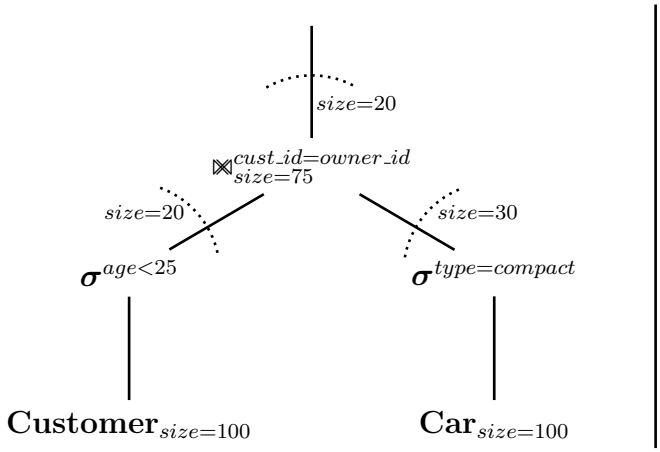
Systematic approaches While sampling approaches aim to produce many queries in an attempt to ensure the DBMS work correctly for a wide range of inputs according to random or guided selection, systematic approaches aim to test the DBMS according to more structured means. One such approach for query generation proposed by Khalek and Khurshid [65] makes use of the formal *Alloy* language – which is designed to model various software structures and constraints [48], as well as to be automatically analysable – and the *Alloy Analyzer* tool to automatically enumerate all possible `SELECT` queries for

a given schema. These queries include additional elements such as extra joins between tables, aggregation operators (e.g., `MAX`, `MIN` and `count`) and subqueries. To use this approach, the tester must provide a description of the schema under test expressed in the Alloy language, where each table is defined as a set of named fields with types and constraints manually defined over them. The tool implementing this approach can then automatically provide a set of queries giving complete coverage of that schema with respect to an SQL grammar specified, which can be varied to determine the range of queries to be generated. While a simple example is shown to execute in a reasonable time using this approach, it may be susceptible to scaling issues as the complexity of the schema and supported grammar increase, because it fully enumerates all queries possible with the given grammar. For example, a simple two table database schema and relatively simple grammar produced over 27,000 queries and while these only took ∼2 minutes to generate, the time taken to execute each of these with a DBMS may quickly become infeasible as the complexity of the schema and grammar increases. It is also unclear how the time taken for the Alloy Analyzer to produce data that meets any defined constraints may scale.

Data generation

In addition to the generation of queries, testing of a DBMS requires a number of rows of data to be present in a database otherwise most, if not all, queries executed during testing would return empty results, preventing much of the DBMS being tested thoroughly. The data generation techniques are divided below into 'query-oriented' [14, 15, 8, 9, 64] and 'schema-oriented' [16, 46] approaches. The query-oriented techniques produce rows with prior knowledge of one or more queries that will be executed against the database, ensuring that data will be returned for each, while the schema-oriented techniques focus on particular structures or relationships in the schema used to define the database.

Query-oriented approaches The *QAGen* data generator [14, 15] produces data that is distributed according to cardinality constraints defined for a given query. These constraints are defined at each branching point of the parse tree for a given SQL `SELECT` statement, and specify how many rows in the produced data should be contained in the result of that branch. A simple example of this is shown in Figure 2.3, where the number of rows to be produced are included at each part of the query. This approach allows




```
SELECT * FROM Customer, Car
WHERE Customer.cust_id = Car.
     owner_id
AND Customer.age < 25
AND Car.size = `compact'
```

Figure 2.3: Example of cardinality constraints over the branches of a query, as used by Binnig et al. [14] to specify what data should be generated. The corresponding database would contain rows for 20 customers aged under 25 who own compact cars, out of a total 100 customers, 20 of whom are under 25, and 100 cars, 30 of which are compact.

the tester to specify the data they wish to be returned by a query, enabling them to target particular areas of interest in the DBMS as well as model realistic distributions of data. The row counts are used to build an internal symbolic database that describes the constraints that generated data must satisfy, similar to techniques used for symbolic execution of programs. A data instantiation phase then applies a constraint solver, *Cogent*, in an attempt to produce concrete instantiations for each row in the symbolic database, creating real data values that can be used in a traditional relational database. To evaluate the *QAGen* tool, Binnig et al. [14] used it to generate data for a database extracted from a DBMS benchmarking system based on cardinality constraints specified for five different queries. This demonstrated that the tool was able to successfully satisfy a complex set of constraints – with multiple SQL joins, many branching points and subqueries – while producing 10 megabytes of data for each in ∼14 minutes. They also showed that the technique could be scaled up to produce datasets of up to 1 gigabyte with a linear increase in cost, allowing it to be applied to large scale data generation problems.

Cardinality constraints were also used by Arasu et al. [8, 9] to describe data generation requirements in a declarative way, implemented in the *DataSynth* tool. In this case, the constraints can describe both intra-table and inter-table column relationships, and define how many rows should be produced for each part of a query, as with QAGen. Additional limitations may also be placed upon certain column values to ensure they contain sensible

SQL

```
CREATE TABLE customer (
  cust_id int,
  cust_name varchar(50),
  PRIMARY KEY (cust_id)
);
```

Alloy

```
1  one sig customer {
2    rows : Int -> varchar
3  } {
4    all x: rows.varchar | one x.rows
5  }
```

Figure 2.4: An example of a relational schema expressed in both SQL and as an Alloy representation, as used by Khalek et al. [64]. Line 2 of the Alloy code represents the `cust_id` and `cust_name` fields as an integer and variable length character, respectively. The `PRIMARY KEY` constraint is represented on line 4, which states the restriction that for all rows there must only be one value for each unique integer value.

values, for example limiting an integer 'age' column to positive values below a reasonable maximum limit. In contrast to the work of Binnig et al. [14], the technique implemented in the DataSynth tool allows one database to be created that attempts to satisfy the constraints for multiple queries with cardinality constraints simultaneously, rather than being limited to producing one database for each query. In addition, DataSynth expresses the constraints as a series of linear equations that are solved using a commercially available linear program solver, as opposed to the more general constraint solver used in the QAGen tool. When applied to the linear equations, the solver produces a probability distribution that defines how likely it is a value should meet particular constraints. This is then sampled from to determine what data should be generated, giving an estimate of the distribution of data specified in the constraints. As a consequence, the approach used in the DataSynth tool is not exact, and instead produces data similar to that which is desired – however, as the number of rows produced is increased, the inherent randomness it applies should cause the number of rows to tend to the specified number. This inexactness allows the data generation process to be much quicker than for more exact techniques, such as the constraint-solving approach used in QAGen, as the time taken to resolve the constraints does not have to scale as the target amount of data does. For example, while resolving the constraint system with constraint solving for a 1 gigabyte data generation problem using the QAGen tool took over 10 minutes, a similar problem expressed using linear equations in the DataSynth tool was solved in less than 5 seconds. While the data generated is only an approximation of those specified by the cardinality constraints, an example data generation problem with ∼9 million rows contained only 264 additional database rows, demonstrating that the amount of error from this approximation would be acceptable for the vast majority of uses.

By modelling a schema and query formally using the Alloy relational language in a framework they referred to as *ADUSA*, Khalek et al. [64] were able to generate test data using the Alloy Analyzer. This tool is able to check whether a set of constraints can be satisfied and then attempts to produce data meeting those constraints, if possible. An example model of a schema using the Alloy language is shown in Figure 2.4. In ADUSA, a query was modelled as a composition of functions – such as `SELECT`, `WHERE` and `FROM` – and relational predicates. This is then used to partition the possible rows of data into those that satisfy a particular part of the query and those that do not. However, due to the expense of the model used for strings other operations cannot be included, preventing the approach being used to test queries containing predicates that use these. For example, a constraint such as `age > 25` would partition the possible data rows into `age > 25`, satisfying the expression, and `age` $\leq$ `25`, failing to satisfy it. The same partitioning in ADUSA can also be applied to string equality, where `name = 'John'` creates partitions where `name` either exactly matches or is any other string. The Alloy Analyzer is then used to enumerate combinations of matching data values that satisfy each predicate, generating rows for a test database as a consequence. The *query-aware* nature of this technique therefore ensures that at least some of the data produced should be returned when the query is executed, rather than the query result being empty. This is important, as otherwise faults in the part(s) of the DBMS handling the returning of the correct results may remain undetected during testing. In addition to producing test data, the expected result of applying the query against the generated database is produced, enabling the testing of a DBMS without requiring an additional DBMS to compare against, as required by some other approaches, or another form of test oracle. An empirical evaluation showed that ADUSA was able to produce data and expected results for a small 5 table schema, containing multiple foreign key constraints, from a set of 4 different queries in at most 78 seconds. However, these queries contained only single clause constraints in the `WHERE` section, suggesting more complex constraint systems may be much more expensive to process. In addition, as the required number of rows increased slightly the execution time did so significantly. This is because Alloy Analyzer enumerates all possible test databases for a given specification, therefore favouring the production of many small test databases that separately test the DBMS with different queries. Further evaluation did however demonstrate that the tool was able to locate a number of known bugs and seeded bugs in commercial DBMSs, suggesting it may still be useful in certain settings.

Schema-oriented approaches Rather than producing test data according to one or more queries, schema-oriented approaches instead makes use only of information in the schema to generate rows of data, avoiding the need for a priori knowledge of the queries being used. For example, Bruno and Chaudhuri [16] proposed an approach for creating *flexible database generators* that make use of an annotated version of the schema to create test data. These annotations specify for each column how values should be generated according to iterating sequences and statistically distributed values – either uniform or normal – that may be combined to produce arbitrarily complex row values. Integrity constraints between tables can be supported by generating data for multiple tables simultaneously, ensuring `FOREIGN KEY` references are valid. Once a tester has specified a statistical model that describes the desired data using these annotations, it is possible to create a variable sized database by retrieving values from the iterators and chosen distributions, which will then approximate the desired data. This can then be inserted into a database so they can be queried during DBMS testing, or also used to benchmark the performance of a DBMS for potentially large datasets. An evaluation of this approach showed it was able to produce a large number of rows quickly, generating 10.5 million rows (∼1GB of data) in approximately 13 minutes. This demonstrates the technique is more scalable than other alternatives, although it does not guarantee that the data will return non-empty query results during testing – this responsibility is instead passed to the tester, who must carefully define the desired column values to achieve this.

The relationships between various tables and columns in a schema can also be modelled as a graph, which can then be used for data generation. Houkjær et al. [46] proposed that each table and inter-table constraint could be represented as nodes and edges in a graph, respectively. Each table node stores information about the types of the columns, desired number of rows (expressed as the proportional number of rows on a per table basis) and any `PRIMARY KEY` or `UNIQUE` constraints. The edges between nodes describe both the type and direction of each inter-table relationship (i.e., which table refers to values in the other), as well as the levels of reference participation (i.e., what proportion of rows in `a` should be referenced to by rows in `b`, where `b` has a foreign key reference to `a`) that must be maintained when creating data. By traversing the graph it is possible to determine which rows of data must be created first such that foreign key relationships can be satisfied by maintaining information about existing rows. However, it is unclear what data generation technique is used by Houkjær et al. to create the data values themselves, meaning it is difficult to replicate either the approach or these empirical

results. Nonetheless, the graph-traversal approach was shown to be able to produce 1GB of data that satisfied multiple intra- and inter-column constraints within approximately 10 minutes, with a linear rise in time taken to increase the amount of data generated, demonstrating both its effectiveness and scalability. As such, it may be a useful approach for generating realistic data for testing the behaviour of a DBMS, especially that relating to inter-table constraints.

2.2.2 Database interaction testing

In addition to testing that the behaviour of a DBMS is as expected, it is important to also test whether an application that stores data using a database (hereinafter referred to as a *database application*) works correctly. Testing a database application is more challenging than testing a traditional application that doesn't make use of a database, as it is more difficult to manage and observe the state – while the state of a traditional application can be observed as the values of one or more variables after a series of method invocations, a database application has additional state information stored in the database. As the state of the database usually determines the behaviour of a database application to some degree, it is therefore important to both populate the database using data generation techniques, as with DBMS testing, and examine or alter this state, by executing queries against it during testing.

The remainder of this Section describes a collection of approaches for creating queries to help identify possible faults in an application, then discusses a number of techniques for generating database queries, both for use in testing of database applications. Finally, a number of optimisation techniques specific to this testing problem are discussed, which help reduce the possibly large cost of testing database applications.

Query generation

Similarly to DBMS testing, it is important when testing a database application to produce queries that may be able to highlight where mistakes may have been made in the application – for example, in sections of code where `SELECT` statements are embedded. Faults in theses locations may otherwise manifest as errors during the execution of the application. However, in contrast to DBMS testing all query generation approaches in

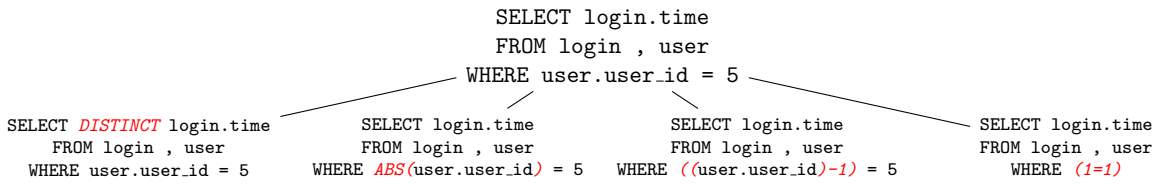


Figure 2.5: An example of 4 out of 12 mutants produced for a simple schema and query using the operators implemented in the *SQLMutation* tool of Tuya et al. [106]. The injected faults are highlighted in each case.

this context can be categorised as 'systematic approaches' [24, 106, 107], designed to test for specific faults that may exist in the application.

Systematic approaches Chan et al. [24] presented a fault-based testing approach using mutation operators to modify queries embedded in database applications, which make use of additional information provided in an enhanced entity-relationship model of the relational schema. This model expresses properties that cannot be included in the SQL syntax of the `SELECT` statement, such as the proportion of rows participating in the relationship that joins them (i.e., a `FOREIGN KEY` constraint) and the specialised class of data in the row (e.g., an `invoice` table may contain both `normal` and `cancelled` classes of invoice). To produce mutants of queries using this information a total of seven mutation operators were proposed. Because the enhanced entity-relationship model provides information about the connections between different tables in the schema, the operators are able to make *semantic* mutations rather than the more common *syntactic* type of mutations[1], which aim to expose business logic errors in the application. These operators injected faults such as altering a column reference in the statement to another of the same data type, replacing a reference to a specialised class with a subclass, and altering the cardinality of a relationship to extreme values. While the semantic mutants produced by this approach may prove useful for testing purposes it is not possible to empirically evaluate this proposition, as the approach has not been implemented in a prototype tool.

The use of mutation to generate `SELECT` queries was also investigated by Tuya et al. [106, 107], who proposed a large number of syntactic mutation operators to model a wide

[1] A semantic mutation involves a small change to the *meaning* of the code whereas a syntactic mutation involves a small change to *syntax* of the code

range of possible faults. These operators included those that apply to the "main SQL clauses" (joins, ordering and predicates), operators in conditions and expressions (logical operators and operations on absolute values), handling of `null` values (alternating `IS NULL` and `IS NOT NULL` statements), and replacement of column identifiers. An example of applying some of these operators is shown in Figure 2.5. Once these mutants have been automatically generated, they can be used to evaluate the quality of test cases and the test data they have created through mutation analysis. The evaluation of test cases involves automatically executing each with the generated mutant queries, considering each to be killed when the mutant returns different data to the original query. In the case that a mutant remains alive, it is either the case that the test database or test case is insufficient to detect the fault, and therefore require further refinement to give confidence that the application will work correctly. A tool based upon this approach, *SQLMutation*, has been made available on-line as both a web application and an XML-based web-service, facilitating possible integration with other work. The mutation analysis approach was applied to evaluate the data supplied as part of the NIST test suite – an SQL conformance suite used to check the correctness of a DBMS according to the SQL standard – which was improved from detecting 70% of the injected faults initially to 85% through automatic data generation, and then to 100% through manually crafted test cases. This shows the tool is able to generate useful data, with respect to the set of faults implemented in the mutation operators. However, it is unclear how well these model the real-world programmer errors that may occur when writing SQL statements as part of a database application. Additionally, the need for manually crafted test cases to detect 15% of injected faults may mean achieving full coverage of faults in a practical setting would incur a significant human-time cost.

Database generation

As the behaviour of a database application is likely to vary depending on the contents of the database it uses, it is important to generate a number of rows of data to insert into the database – otherwise those sections of code only executed when such rows exist cannot be tested, and therefore may contain undetected faults. By manipulating the choice of data values in these rows it is possible to test different parts of the application, ensuring each of these parts behave as expected. The techniques for database generation discussed below are divided into 'flexible-use database generation' [29, 15], which generate

data with multiple possible testing uses, and 'specialised database generation' [39, 98], whose data is generated to detect specific problems.

Flexible-use database generation To generate each row to be used in the test database, de la Riva et al. [29] transformed the queries extracted from the database application into a series of coverage rules that describe the test case, using a coverage criterion named *SQLFpc*. This coverage criterion defines what test data is required to fully test any conditions in the query – for example, a `WHERE` clause of `x` ≤ 5 would create the coverage rules that represent `x < 5`, `x = 5` and `x > 5`. Once a series of coverage rules have been created they are transformed into a representation that uses the Alloy relational language, which are combined with an Alloy encoding of the database schema before being processed using the Alloy Analyzer tool. Each of the coverage rules is then 'solved' using this tool, which generates an example row of data that satisfies that rule. This data is then used as a row in the database, which should contain one row for each coverage rule and therefore ensures each query returns at least one result. Evaluating this approach, the data generated using the SQLFpc coverage criterion was compared against a production database in terms of the number of coverage rules satisfied and the proportion of mutants killed (using mutants generated by Tuya et al. [107]), for a single large relational schema with 37 tables and 230 columns. In terms of coverage the SQLFpc test database achieved an average of 87% compared to 57% for the production database, while mutation score averaged 84% compared to 67%. Therefore, the SQLFpc test database was significantly more effective than the production database at detecting the injected faults. In addition, execution of the SQLFpc test database should be much faster as it contained approximately two orders of magnitude fewer rows (139 vs 139,259), although data relating to this execution cost are not reported by the authors. Overall, the SQLFpc was therefore much more efficient at generating useful test values than using data taken from the production database, suggesting that the realism of the data generated may not be as important as other criteria when creating test databases.

Using techniques referred to as *reverse query processing* (RQP) [13] and subsequently multi-RQP (MRQP) [15], Binnig et al. enable a tester to describe the desired data in terms of the expected results for a series of queries. In general, these queries should refer to results that are independent of one another (i.e., the data generated for one should not affect the results of different query), however MRQP does support some types

of overlap in queries through a mechanism called query refinement. This information is then used in a data generation process which guarantees that when the queries are executed they will return the expected result – although the exact mechanism used for producing data is not clearly detailed. The data can then be used for a variety of purposes such as database application testing and measuring the performance of a given DBMS. Unfortunately, the MRQP approach has not been evaluated in an empirical experiment, with only a demonstration with a small toy example schema being included. In this case, the approach was able to generate a small database according to a set of queries and expected results, however there are no details given on the amount of time or space necessary to generate this. The efficiency of this approach is therefore unclear, and it is also unknown whether it is able to scale well with larger examples, greater numbers of queries or higher required numbers of rows.

Specialised database generation Instead of generating data that may be useful for a variety of purposes, some techniques focus specifically on identifying a certain set of possible faults. For example, the *X-Data* system proposed by Gupta et al. [39] was designed to produce database rows guaranteed to return different results for a subset of join operator and selection predicate mutations, applied to `SELECT` statements. Join operators are used to combine tables, based upon some matching column value, and can vary on whether they include only matching rows, all rows from one table or all rows from both. It is therefore useful to test whether the type of join is correct as otherwise too many or too few rows may be returned in the results. Selection predicates, such as $=$ and $<$, are used in `WHERE` clauses to limit the number of rows returned according to some boolean expression. The X-Data system provides the tester with a small amount of data that they can use to ensure these types of fault would be detected, although the exact means of data generation is not specified. In addition, the tester must supply the expected results for each query, increasing the human-cost of testing database applications in this way. An implementation of the X-Data system was tested with a single sample schema using queries containing multiple joins, which showed data could be produced for these within 10 to 15 seconds, although the exact attributes of the schema are not discussed.

Expanding upon the X-Data system, Shah et al. [98] incorporated additional rules for generating data to kill mutants of aggregation clauses (e.g., `COUNT`). However, the technique for these relies upon a number of assumptions about the schema and queries.

These include primary and foreign keys being the only constraints, as well as disallowing the use of `null` values. In addition, nested sub-queries are also not supported. To generate data to kill a specific mutant, the approach expresses each of the applicable constraints between variables symbolically in a format compatible with the *CVC3* general purpose constraint solver. Applying this solver creates an array of values that can then be transformed back into database rows that are guaranteed to produce different outputs for the non-mutant and mutant queries, thus killing the mutants. By examining whether the database application returns the expected results in each case, as determined by the tester, it is possible to identify possible faults in the supported clauses of `SELECT` statements. Shah et al. also proposed a means of making generated datasets more intuitive for human analysis, potentially reducing the human-cost of determining whether the results are as expected, by using values extracted from an existing database where possible. This is implemented into the data generation process by using an additional constraint that forces the CVC3 solver to use values from rows in the database, provided that there are enough rows present. An experimental evaluation of this technique showed it was able to generate small test databases that were able to kill all of the considered types of mutants, provided that equivalent mutants are removed via other means. However, the nature of this experiment was limited as only seven example queries were tested against a single database schema, limiting the generalisability of the findings.

Optimisation techniques

As the state of a database must be managed during testing, as well as that of the application, testing database applications can become very expensive. To ensure that this cost does not become prohibitive, a number of techniques have been proposed that aim to reduce some of the costs specifically associated with database application testing, two of which are described below.

During testing, if the size of the test database is non-trivial then there is an expensive cost to 'reset' the database to the initial state between executing each of the test cases. This is necessary to prevent a database state-altering query (such as `UPDATE`) causing a subsequent query to return a different result than it otherwise would have. To reduce this cost for the purposes of regression testing, Haftmann et al. [41] proposed a series of algorithms that could be applied to determine which tests can be executed in sequence, without a database reset, while still obtaining the correct results. The first of these

algorithms, *Optimistic++*, executes the tests in a random order, resetting the database and re-running a test only if it fails initially. During subsequent executions, information about conflicts between tests can be used to further reduce the execution time. While this algorithm may suffer false positives – where a test should fail, but does not because of the non-empty database state – the authors claim this is both rare in practice, as well as a potentially acceptable cost for the significant increase in the efficiency of testing. The *Slice* algorithm reduces the number of database resets required by using the test case conflict data as a heuristic to select as many test cases as possible into sequences. As well as reducing the number of database resets, the proposed framework also supports the distributed execution of the test sequences across multiple servers, each with their own database instance, to reduce the overall running time. An empirical evaluation with 1,000 test cases, with execution times of between 0 and 3 minutes, and a varying number of conflicts, ranging from 1,000 to 100,000, showed the reset-reduction algorithms were able to execute a large number of tests while reducing the number of resets significantly. In addition, both parallelism via multiple local threads of execution and multiple separate servers were able to significantly decrease the time further, with little overhead being incurred in the latter case.

Instead of attempting to order the test cases to improve the efficiency of testing, Tuya et al. [108] instead aimed to reduce the size of the test database. The *Query-Aware Shrinking* approach they proposed makes use of automatically generated coverage rules from queries used in the test cases to identify which rows in the existing database are required, to ensure one row of results is returned. Where possible, when choosing a row to add to the minimised database from a set of results returned by a query, a row already required by another query is used to avoid unnecessary additional rows being stored. Once a set of rows have been selected all others are discarded, except those required for referential integrity (i.e., `FOREIGN KEY` constraints). This approach is therefore able to automatically create a test database that still returns at least one row of results for each query, but which likely has significantly fewer rows of data. This in turn reduces the cost of database application testing by decreasing the time taken to insert the rows of data into a database, to reset the database when needed and to execute queries against it. In an empirical evaluation, the approach was applied to the reduction of a production database containing a non-trivial 137,940 rows, based upon a schema with 31 tables. The reduced test database contained only 223 rows – a 99.84% reduction – whilst only taking approximately 2 minutes to produce, likely leading to a significant improvement in the

efficiency of application testing. In addition, a mutation analysis experiment showed that the mutation score achieved by the reduced test database was less than a percent lower than the full database, confirming that the smaller database still retained a very similar fault-finding capability.

2.3 Mutation Analysis

Mutation analysis is a technique that evaluates the quality of a test suite according to whether it is able to detect the presence of a series of automatically injected faults. Where a suite detects all such faults the user can assert, with an adequate degree of certainty, their program is unlikely to contain such mistakes. Mutation analysis can also be used to guide automatic generation of test data (e.g., [37, 44]), although this is considered outside of the scope of this thesis. This Section provides an overview of the application of mutation analysis for software testing, then discusses a number of important topics and issues relating to mutation analysis that provide useful background for later Chapters of this thesis.

2.3.1 Overview

Mutation analysis has commonly been used to compare and evaluate various approaches to creating test cases in software testing literature. It measures the effectiveness of the tests by checking whether they can identify a series of potential programmer mistakes, as modelled by a number of small changes made automatically to a program by applying one or more *mutation operators*. Each version of the program that has had such a change applied is referred to as a *mutant*. A mutant is detected, or *killed*, by a test suite if it exhibits different behaviour when executed with the mutant, compared to the original program. By killing the mutant the test suite enables the tester to assert that the program does not contain the specific mistake modelled by that mutant. Conversely, if the behaviour is the same for both the mutant and original program for all test cases, the mutant is *alive*, meaning the test suite is not sufficient to detect the possible fault contained in that mutant. The tester can then refine the test suite according to the alive mutants, until it is able to kill all of mutants, and is described as *mutation adequate*.

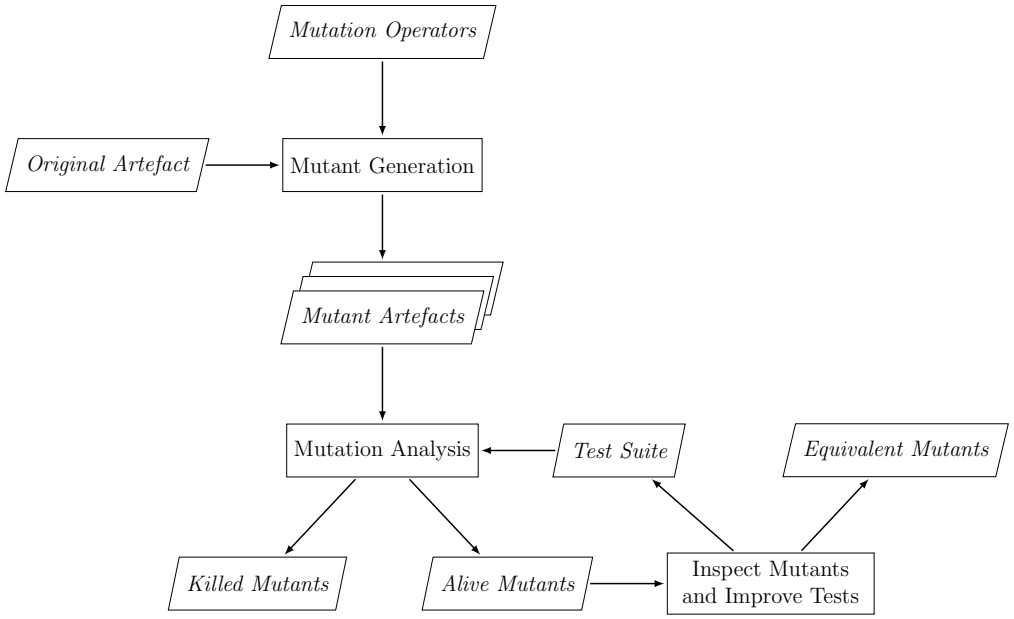


Figure 2.6: An overview of how mutation analysis is used to evaluate the effectiveness of a test suite.

The effectiveness of a test suite is usually measured using the higher-is-better metric of *mutation score*, calculated as *killed mutants / all mutants*, which ranges from 0 to 1. The change in some mutants may have failed to alter the behaviour of the program, and therefore cannot be detected by any test suite. Such mutants are known as being *equivalent* to the original program and usually require manual analysis to identify and remove them. Figure 2.6 provides a summary of how mutation analysis can be applied to determine the effectiveness of a test suite.

2.3.2 Background

The concept of mutation analysis is not a new one; the first practical method for mutation analysis was described in 1978 by DeMillo et al. [30], who referred to their approach as *program mutation*. This aimed to exploit two key concepts – the competent programmer hypothesis and the coupling effect – that DeMillo et al. argued provide a logical basis for program mutation. The *competent programmer hypothesis*, states that programmers will produce programs which are almost correct, suggesting that small syntactic changes to the program should prove sufficient to detect and resolve possible faults. Meanwhile, the

coupling effect, asserts that tests able to detect small errors should be sufficiently sensitive to identify other, larger errors. Therefore, determining the effectiveness of a test suite at identifying a number of small injected faults (i.e., mutations) will give a reasonable measure of effectiveness at detecting a much broader set of faults. This process has since become known as mutation analysis, following a similar process as DeMillo et al. [30], as described in Section 2.3.1.

Since its initial inception, mutation analysis has attracted significant interest, with a 2011 survey revealing that by 2009 over 400 related works had been published [52]. The remainder of this Section now discusses a number of these publications in context of a number of key issues faced when applying mutation analysis. Section 2.3.3 describes attempts to show that both the coupling effect and competent programmer hypothesis, which underpin the theory of mutation analysis as a testing technique, are valid assumptions. Section 2.3.4 explains the problem of equivalent mutants, which cannot be killed by any test case, and how they may be identified. Section 2.3.5 discusses the potentially large computational cost of mutation analysis and techniques that allow this to be mitigated.

2.3.3 Underlying concepts

Coupling effect

To experimentally evaluate the claim of the coupling effect, Offutt [82, 83] empirically tested the similar *mutation coupling effect*. This states that complex mutants and simple mutants are coupled, so a test suite that can kill all simple mutants of a program will also kill most complex mutants. Assuming that the faults injected by the mutation operators used to produce these mutants are representative of real-world faults, showing the mutation coupling effect to exist also supports the theory of the general coupling effect. The experiment made use of the Mothra mutation analysis system for Fortran [66], utilising 22 mutation operators believed to be representative of real-world programmer mistakes. The simple mutants were created by applying one mutation operator, while complex mutants were created by applying two or three operators (these are known as second-order and third-order mutants, respectively, and are collectively known as *higher order mutants*). While the programs studied in this experiment were necessarily small

(due to the computational cost), the results showed strong support for the mutation coupling effect, with over 99% of non-equivalent complex mutants being killed by data sets generated to kill all of the simple mutants. In addition, the experiment showed that data sets which killed a small proportion of simple mutants were still able to kill more complex mutants. For example, in one case tests killing 75% of simple mutants were able to kill 92% of complex ones. Similar results were also found by Lipton et al. [71], although they only considered a small proportion of the mutants produced, thus limiting the generalisability of their findings. In both cases, these results provide supporting evidence for the coupling effect.

The coupling effect has also been examined with a more theoretical approach. Wah [113, 114] modelled programs as compositions of functions, where a fault is limited to affecting one of these functions and each fault in a higher order mutant relates to a different function. Their conclusions supported those of the empirical evaluations, showing that a test suite which is adequate for simple mutants will kill an increasing number of higher order mutants as the order is increased – that is, it will kill more third-order mutants than second-order, and more second-order than first-order. In addition, Wah showed that the more simple mutants each test case is able to kill, the greater the number of higher-order mutants the whole test suite will identify. This was complemented by Kapoor [63], who demonstrated a formal proof that the coupling effect holds for eight different types of faults in boolean formula (as used in conditional statements) – including an omitted conjunction or disjunction, an incorrect choice of operator, a wrong operator reference, an operator precedence issue due to missing brackets, and replacing a variable reference with both 0 and 1. Together, these provide a theoretical backing for the coupling effect, which in turn provides confidence in the validity of mutation analysis as a testing technique.

Competent programmer hypothesis

When describing the initial methodology for mutation analysis, DeMillo et al. [30] also reported on the proportion of different types of mistakes made by programmers when writing programs in Fortran, Cobol, PL/I and Basic. Although errors made with these languages may not be directly comparable to those found in programs using modern languages, the results showed the majority of errors were caused by simple mistakes, therefore supporting the competent programmer hypothesis. Acree et al. [2] also argued in support of the hypothesis, giving an example of how the approach to a problem might

differ between a competent and incompetent programmer. In the latter case, Acree et al. asserted that errors of such magnitude would be trivially easy to identify with basic test cases, and therefore needn't be considered when performing mutation analysis.

Although not directly referring to the competent programmer hypothesis, numerous examples from the published literature have commented on the types of errors caused by mistakes in programs, which provide additional confidence in the correctness of the hypothesis. Analysing faults made when writing an assembly program of approximately 4,000 lines, as self-reported by programmers, Shooman et al. [99] also showed that the majority of errors were fixed by a change to an instruction – a relatively small change. Notably, of the 63 reported fixes included in the study, none required algorithmic changes, further supporting the assumption that programmers generally only make subtle faults. Budd et al. [19] discussed both the theoretical basis for mutation analysis and the classified 25 errors in 11 Fortran programs, revealing that almost half related to an incorrect computation statement. This, again, suggests that many programmer mistakes are simple in nature, and therefore that the competent programmer hypothesis appears to hold true. The types of errors found in an industrial setting were investigated by Ostrand et al. [90], where programmers were required to record information about errors during development of a single application. Their results showed that the majority of errors were caused by mistakes in code handling constant values and code that initialised or modified variable values, representing 32% and 22% of cases respectively. Interestingly, the experiment also involved recording the time taken to isolate the error and then fix it. These results showed that in 71% of cases neither activity took more than 1 hour, suggesting the mistakes causing them were relatively small. In summary, each of these examples from the published literature provide some support for the competent programmer hypothesis, giving additional confidence that mutation analysis provides a suitable means of detecting faults.

2.3.4 Equivalent mutant problem

Overview

In general, it is intuitive to believe that applying a mutation operator to a program, and thus changing it syntactically, should always result in a change in behaviour – even if this is relatively small, or only revealed when using highly specific input data. However,

Original

```
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Equivalent mutant

```
int max(int a, int b) {
    if (a >= b) {
        return a;
    } else {
        return b;
    }
}
```

Figure 2.7: A function with an example equivalent mutant.

as described in Section 2.3.1, this is not always true; it may be the case that no possible input to the program causes a mutant to return a different result than the original. Such mutants are known as *equivalent* mutants. Figure 2.7 provides an example of a function, which returns the maximum of its two arguments, and an equivalent mutant that could be produced via a small syntactic mutation. By definition, equivalent mutants cannot be killed by a test suite, therefore their presence during mutation analysis causes a decrease in the mutation score, by increasing the proportion of live mutants to killed mutants. If equivalent mutants are not identified and removed then the tester may be led to believe that their test suite is insufficient to reveal additional possible faults in their program, as the mutation score indicates it is not mutation adequate. It is therefore desirable to identify and remove equivalent mutants, to ensure that the mutation score provides a more accurate estimate of the fault-finding capability of any given test suite.

While equivalent mutants may be identified by hand, the potentially huge number of mutants produced means this can quickly become infeasible for all but simple programs, due the both the human time cost and the risk of errors. Unfortunately, deciding whether a mutant is equivalent to the original program, commonly known as the *equivalent mutant problem*, has been shown to be undecidable [18]. Consequently, detection of all such mutants using automated techniques is not possible.

Proposed techniques

Although the equivalent mutant problem is undecidable in the general case, there have been numerous techniques proposed to automatically identify at least some of the equivalent mutants of a program, therefore reducing the human cost associated with mutation analysis. The remainder of this Section discusses and evaluates a number of these techniques.

Offutt and Craft [85] attempted to exploit the intuition that equivalent mutants are often effectively optimised or de-optimised versions of the original program – that is, they are semantically identical, but their syntactic differences may be produced by applying or removing optimisations that may be made by a compiler. The paper introduced a total of six techniques for detecting equivalent mutants of Fortran programs in a tool integrated into the Mothra mutation system. Dead code detection uses control flow analysis to statically determine when a particular statement cannot be executed. If the mutation affects code that can be proved to be unreachable, then the mutation cannot affect the program behaviour and the mutant must be equivalent. Constant propagation detects when the specific value of a constant means certain mutations have no impact. For example, a mutation taking the absolute value of a positive number produces the same number, thus execution of the program will be unchanged. Similarly, if the value of some variable is known to be constant then a mutation setting the variable to that value will produce an equivalent mutant (e.g., if data flow analysis shows `a = 0` then the mutant produced by an operator setting variables to `0` will be equivalent). Invariant propagation stores information about the relationships between variables and 0 (e.g., `x > 0` or `y >= 0`), which can be used similarly to constant propagation to detect cases such as when absolute value mutations will not change the program semantics (i.e., when `var > 0`). Common subexpression detection identifies when two variables are being set according to the same expressions, and substitutes a variable reference in place of repeating the calculations of the expression. This is used in equivalence detection by adding to the collection of invariants used by the other techniques. Loop invariant detection is usually applied by a compiler to move code from the inside to the outside of a loop. One of the mutation operators applied in the experiment moved the boundary of a loop, either adding or removing a line from the inside of it. If the statement moved leads to an invariant, then that mutant is equivalent, and can be detected in this way. Finally, hoisting and sinking of statements involves moving statements that are repeated within a particular block of code (i.e., within both branches of an `if else` construct) to remove the duplication. This can detect mutants that are produced by moving the bounds of these blocks. The execution time for each of these techniques is either quadratic or cubic of the program size, however given the full test suite would otherwise be executed for each equivalent mutant (because no test case will kill them), they argue that this time is comparatively very small. To evaluate their tool, Offutt and Craft used Mothra to produce mutants for 15 Fortran programs of 5 to 52 lines, yielding between 180 and 3000 mutants, where the

equivalent mutants had previously been identified by hand. The results showed that of the six techniques, only three were effective in detecting equivalent mutants – dead code detection, constant propagation and invariant propagation. Overall, 45% of equivalent mutants were successfully identified, with a strong bias towards operators specifically targeted by the techniques, leaving 141 equivalent mutants undetected. For the most common type of mutant (`ABS`, which surrounds expressions with a positive and negative absolute operation), which replaces variables with positive and negative absolutes, 59% of the equivalents were identified, whilst none of the equivalent mutants produced by four of the less common operators were detected, representing 11% of the overall total. While these results represent a significant reduction in the number of equivalent mutants that must be hand-checked for equivalence, the quantity remaining for more complex programs (greater than the 52 lines in this experiment) may still present too great a cost for practical usage. In addition, it may be that outside the context of Fortran programs, these techniques may not be as applicable.

Offutt and Pan [86] made use of *feasible path analysis* – determining whether a particular path of execution through a program is possible – to try to identify whether there is input data that is able to kill a mutant. In the case that there is not, they posit that the mutant is equivalent. This is described as the *feasible test problem*, whereby given the requirement for a test case (i.e., to kill the mutant), one must identify if any possible input data exists to satisfy this. To kill a mutant, the test case must satisfy three characteristics – reachability, necessity and sufficiency. These state that the test must execute the mutated statement, that the statement changes the state of the program, and that state must differ at the end of execution, respectively. The requirements for the test case are formed as a constraint, which may be augmented with additional user-specified information such as preconditions on functions, that is then tested for infeasibility. Although identifying equivalent mutants in this way is said to be generally undecidable, Offutt and Pan made use of a collection of specialised techniques and heuristics to determine when the constraints are infeasible, using domain knowledge of the mutation operators. These were implemented in a prototype tool, Equivalencer, which was evaluated against 11 Fortran programs of 11 to 30 statements, producing between 180 and 3000 mutants. The equivalent mutants had been identified manually for each of these programs as part of previous work. The results showed that, on average across all 11 programs, 47% of equivalent mutants could be identified automatically, ranging from 13% in the worst case to 84% in the best. As with previously discussed techniques, while this technique de-

```
1 printMinMax(array) {
2   min = array[0]
3   max = array[0]
4   for (i=1,i<size(array),i++) {
5     if (array[i] < min)
6       min = array[i]
7     if (array[i] > max)
8       max = array[i]
9   }
10  print(min)
11  print(max)
12 }
```

(a) Original program

```
min = array[0]

for (i=1,i<size(array),i++) {
  if (array[i] < min)
    min = array[i]


}
```

(b) Slice for `min` to line 9

```
max = array[0]
for (i=1,i<size(array),i++) {


  if (array[i] > max)
    max = array[i]
}
```

(c) Slice for `max` to line 9

Figure 2.8: An example of program slicing, derived from Hierons et al. [45].

tects a large number of equivalent mutants, it is unclear how well the technique could be generalised to other programming languages and how well it scales with respect to more realistic sizes of program. In addition, the authors state that the tool's implementation is closely coupled with a data generation technique, which may mean application of this approach for other languages could have considerable implementation costs.

Hierons et al. [45] applied program slicing to the equivalent mutant problem, attempting to both simplify the process of human analysis of equivalent mutants and also reduce the number of equivalent mutants produced initially. Program slicing is a technique that produces a reduced version of a program by including only those statements relating to a specified variable, up to a given statement number. Figure 2.8 provides a simple example of program slicing for an imperative programming language, derived from an example given by Hierons et al. [45]. They showed that program slicing may be helpful in simplifying an alive mutant presented to a tester, reducing the complexity of the program they are required to analyse to check for equivalence. This may be combined with a constraint solving approach that can be applied first in an attempt to detect equivalence, or generate a test case able to kill the mutant. The combination of these techniques reduces the number of mutants the tester must consider, thus reducing the overall human cost of mutation analysis. In addition, program slicing is shown to reveal some cases where the mutation cannot affect the overall behaviour of the program, and is therefore equivalent. However, it is not shown how this technique may perform in a practical setting, and therefore how much human effort is saved. In addition, no empirical analysis is given to quantify how many, and what proportion of, equivalent mutants can be automatically identified and at what computational cost.

Harman et al. [43] investigated how dependence analysis could be used to prevent equivalent mutants from being produced and to generate test data for killing non-equivalent mutants. Dependence analysis provides information about which variables affect the value of another variable, at a given line – known as the variables it *depends* upon. This provides a finer-grained understanding of the flow of data in the program than that given by program slicing, such as that used by Hierons et al. [45]. By extracting which variables others depend upon, it is possible to identify which input values may be able to influence the behaviour of the program for a specific mutation. This can be used to reduce the scope of the input domain to consider when formulating a test case aiming to kill a particular mutant. In addition, if for a particular point in the program (the end of the program for traditional mutation analysis) the program variables do not depend on those on the left hand-side of the mutated statement, then the mutation cannot affect the program behaviour. As such, these mutants can be classified as equivalent. Although this technique may help reduce the number of equivalent mutants, Harman et al. state that due to "imprecision inherent in dependence analysis" some proportion of equivalent mutants may not be detected in this fashion. Therefore, further work is required to evaluate the effectiveness, as well as the efficiency, of this approach.

Adamopoulos et al. [3] proposed using a co-evolutionary search technique with genetic algorithms to simultaneously evolve both a set of mutants and test cases for each. This aims to reduce both the number of equivalent mutants and the total number of mutants. An evolutionary search technique attempts to find a solution to a problem by ranking possible solutions according to some fitness function and then combining parts of high ranking solutions, according to operations inspired by biological evolution. To provide meaningful guidance to this process, the fitness function must provide a score encoding how close each solution is to successfully solving the problem, which is then either minimised or maximised (which of these is generally an implementation detail) by the search algorithm. Such a technique was first used to produce a set of mutants which are hard to kill, but are nonetheless not equivalent, according to a fixed set of test cases. This required a fitness function that scores each mutant based on how easily it is killed by the tests, where scores of near 1 indicate a hard to kill mutant, but mutants who are not killed are given a score of 0 to prevent the inclusion of possibly equivalent mutants. Next, an evolutionary technique was applied to select test cases, from a randomly generated pool of tests, according to how effective they were at killing mutants, taken from a fixed pool. These two techniques are then combined to co-evolve both a set of mutants and

a set of test cases for killing those mutants. An analysis of applying these techniques with simulated fitness function values showed that genetic algorithms appear to be well suited to this task. However, because all of the mutants selected are guaranteed to be non-equivalent, it is possible that some hard to kill non-equivalent mutants may be given very low fitness scores, and thus be excluded. These may represent difficult-to-detect faults, which the tester may otherwise fail to identify before they manifest as failures. In addition, efficiency of the technique is not discussed, leaving an open question as to the practical feasibility of the technique.

Fraser and Wotawa [36] explored how mutants of formal program specifications could be analysed to identify equivalent mutants, without the need to use a model-checker to test each mutant against the original specification for a difference in behaviour, as with prior work. This formed part of a wider work where the number of test cases produced and the time taken to generate them was reduced by removing equivalent mutants before test case generation, as well as attempting to identify where tests kill multiple mutants and therefore do not need duplicating. To achieve this, *characteristic properties* of the mutant models are identified, which describe how the mutation made either adds to the set of valid transitions in the program – a formal description of how the state of the program can be changed – or removes an element from it. From this, an equivalent mutant can be identified as a mutated specification where the set of transitions is unchanged, such that no input during the execution of the program could lead to a different state when executed with the mutant than the original. In an empirical study, this approach was shown to greatly reduce both the test suite size and test generation time when applied to one example program, by an average of 77% and 64% across the mutation operators used, respectively. However, detailed results relating to equivalent mutants were not reported.

Grun et al. [38] made initial investigations into whether the impact of a mutant – how much it alters the behaviour of a original program – can be used to predict whether it is equivalent. The changes in behaviour between the mutant and the original program are measured according to the control flow, assuming that if this is changed then the mutant is less likely to be equivalent. The control flow is measured according to the statement coverage achieved by the test suite with each program, recording how many times each statement has been executed. This approach was implemented as an extension to the "Javalanche" mutation tool. The approach was evaluated by examining a sample of the 9,819 mutants produced for the JAXEN XPATH engine – an open-source Java XPATH

query system. An initial sample of 20 mutants was taken from those 1,933 mutants which were covered by the test suite (i.e., the mutated statement was executed). These 20 mutants were manually examined by two experienced individuals who classified each as non-equivalent (proven by providing a test case that killed it), equivalent or undecided. Overall, a mutant was deemed non-equivalent if either individual provided a test case that could kill it, equivalent if both individuals specified so, or otherwise undecided. This human analysis revealed that of the 20 sampled mutants, 10 (50%) were non-equivalent, 8 (40%) were considered equivalent, and 2 (10%) being left undecided. On average this analysis took 15 minutes per mutant, highlighting the high cost associated with human identification of equivalent mutants. Of these mutants, 6 out of 8 equivalent mutants were found to have no impact, while 6 out of 10 non-equivalent mutants did have an impact. These results suggest that a mutant which does have an impact on execution is more likely to be non-equivalent than equivalent. Analysis of those mutants that instead were killed by the test suite showed that 98% had an impact on coverage, supporting the conclusion that there is a link between the impact of a mutant and whether it can be killed. All mutants not killed by the test suite were then ranked according to their impact, with the first and last 20 mutants being hand-examined, according to the same strategy as before. Of the 20 mutants with the highest impact, 18 were non-equivalent and 2 equivalent, while for the 20 mutants with the lowest impact, 9 were non-equivalent and 11 equivalent. These findings also support the hypothesis that there is a correlation between the impact of a mutant and whether it can be killed, with higher impact increasing the chance. Although these results help suggest which mutants are more or less likely to be equivalent, this cannot be used to definitively identify these mutants – only predict, with a statistically significant probability, the most likely classification. However, this may still be useful in a practical setting, to guide the tester to produce test cases for mutants which are most likely to be killed. Ranking the mutants by impact also helps the tester identify where omitted tests could identify potential bugs in many parts of their program.

Schuler and Zeller [96] examined how coverage information gathered at runtime can be used to calculate the likeliness of a mutant being non-equivalent, expanding upon the previous work of Grun et al. [38]. While test cases effectively assess the impact of each mutant at the end of execution (i.e., whether the overall result of the program has changed), this approach involves measuring how the program state has been changed during execution. This impact is measured according to two metrics – control flow and data. The control flow metric measures whether the statements in a program have been

executed in a different order for the mutant than the original program. If the control flow does change, it is more likely that the mutation has propagated and caused an overall change in behaviour. The data metric considers whether the values passed between different methods called during the tests are different. Again, if these have changed then the mutant is more likely to have altered the overall behaviour of the program. To evaluate their approach, seven real-world Java programs were selected, ranging from 4,837 to 94,902 lines of code, producing a total of 116,551 mutants. From these mutants, 20 that were not detected by the test suite – and therefore are either equivalent, or non-equivalent and the test suite is insufficient to kill – were selected per program, giving 140 mutants to manually classify. The results showed that if a mutation does impact upon the coverage it has a 75% chance of being non-equivalent. This suggests that by ranking mutants by coverage may be an effective way to present alive mutants to a tester, reducing the likelihood that the mutant they are attempting to kill is equivalent until the majority of remaining mutants are equivalent. While these results were obtained for large, real-world programs, they are limited to detecting coverage only when the provided test suite executes the mutated statement. This constrains the number of mutants that made up the sample 20 used per program, and therefore possibly limits the generalisability of the results. In addition, the seven programs used may not be representative of other Java programs, especially as they all belong to open source projects, which may have significant differences to programs developed in other settings.

Yao et al. [119] manually investigated the causes and prevalence of equivalent and stubborn mutants (those which are not killed by otherwise effective test suites, but are not equivalent). A total of 1,230 mutants that were not killed by test suites covering all branches from 18 programs were examined, revealing 946 equivalent mutants. These were then classified by whether they were caused due to issues with *reachability* (can the mutated statement be executed at all), *infection* (does the mutated statement cause a change in state) or *propagation* (can the state change affect the program output). For a mutant to be detected, each of these must be satisfied in order – that is, without reachability, infection cannot be achieved, and in turn without infection, propagation is not possible. In-depth analysis of the equivalent mutants revealed that the mutated statement was reachable for the vast majority of cases – a total of 92% – with similar proportions from a lack of infection or propagation – 52% and 40% of all equivalent mutants, respectively. In addition, over 75% of equivalent mutants were shown to be produced by two mutations of the five applied mutation operators – absolute value insertion (ABS, surround variables

with a positive and negative absolute operation) and unary operator insertion (UOI, replace variables with pre- and post- increment and decrement operations). From their results, Yao et al. suggested that the high proportion of equivalent mutants produced by the ABS operator mean it should not be used, unless there is specific reason to suspect the presence of such errors. In contrast, while the UOI operator as a whole led to many equivalent mutants, these were mostly isolated to the pre- operations. As such, it may be useful to split this operator and apply only this part.

Papadakis et al. [93] expanded upon the concept of using compiler optimisations to detect equivalence, applying it to C programs by using modern compiler optimisations. While previous work by Offutt and Craft [85] made use of optimisations inspired by those implemented in optimising compilers, Papadakis et al. [93] compiles the original and mutant programs using an existing compiler with these optimisations enabled then compares the machine code produced. While these optimisations may alter parts of the program, generally to reduce the cost of execution, they do so in such a way that the behaviour is unchanged. Therefore, if two programs compile to the same optimised machine code, they must be equivalent[2]. In addition, the same approach is taken to detect *duplicate mutants* – those that are equivalent to other mutants and therefore represent the same possible programmer error, such that any test that kills one always kills the other, and vice-versa. This technique is referred to as *trivial compiler equivalence*. To evaluate their approach, Papadakis et al. used the popular `gcc` compiler with four different optimisation settings ("None", "`O`", "`O2`" and "`O3`", in increasing order) to find equivalent mutants for two different sets of programs. The first consists of the two largest components extracted from each of six large scale real-world programs, which vary from 7,323 to 750,157 lines of code (although the components themselves range from 1,075 to 22,827). The second contains 18 programs which have previously been manually analysed to identify equivalent mutants, used to provide a "gold standard" number of equivalent mutants with which to compare the approach. The results for the first set of programs revealed 9,551 equivalent mutants and 27,163 redundant mutants, representing 7.4% and 21.0% of all mutants. Interestingly, the highest optimisation setting, `O3`, only detects all equivalent mutants that the lower levels of optimisation do in 2 of 12 cases, although on average it identifies 84% of equivalent mutants, and for 8 of 12 programs it does detect the greatest proportion. These results suggest that sometimes mutations

[2]Alternatively, there may be a bug in the compiler optimisations. However, the maturity of C compilers significantly reduce this threat, such that it can generally be ignored.

may prevent some levels of optimisations being applied by the compiler, thus to detect as many equivalent mutants as possible using this technique may necessitate multiple compilations and comparisons, with each different optimisation level. In contrast, when attempting to detect duplicate mutants the `O3` level identifies almost all of the 27,163 duplicate mutants, recognising all of those found by other optimisation levels for 10 of 12 programs, and identifying almost all of them in the other two cases – 97% of them in the worst case. In terms of computational cost, the technique required 377,166 seconds (∼105 hours) to compile all 12 of the first set of programs with the `O3` optimisation, detect equivalent mutants and detect duplicate mutants. While this seems very expensive, the compilation step that represents over 99% of this time is already required to execute the test cases for mutation analysis, although compilation without optimisations could reduce the time to 71,301 seconds (∼20 hours). In addition, given the total of 129,161 mutants this represents less than 3 seconds per mutant, which is clearly faster than the alternative of human analysis, and is a purely computational cost, which may be further reduced by utilising additional hardware resources. The second set of programs was used to evaluate what proportion of known equivalent mutants the trivial compiler equivalence technique is able to detect. These results show that the technique detects between 9% and 100% of equivalent mutants, with an average across programs of 30%. Unfortunately, the mutant programs had not been analysed for duplicate mutants, so the proportion of these detected could not be evaluated. Regardless, this confirms that the technique can successfully identify a significant number of equivalent mutants using checks which cost relatively little – especially compared to human analysis. Overall, trivial compiler equivalence provides a technique for detecting both equivalent and duplicate mutants that seems both scalable and effective. It is, however, reliant on a sufficiently effective and reliable optimising compiler being available for the programming language used in the program of interest, as well as a tool to compare the resulting executable files. In addition, it may be necessary to compile programs with multiple optimisation options, if available, to reveal the highest proportion of equivalent mutants, increasing the computational cost of mutant compilation prior to mutation analysis.

2.3.5 Execution cost

As described in Section 2.3.1, mutation analysis requires the test suite to be executed once for each mutant. Given the number of mutants increases as the size of the program does,

Original

```
1 int max(int a, int b) {
2     if (a > b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
```

Meta-mutant

```
1 int max(int a, int b) {
2     if (relop(absvar(a,2),absvar(b,2),2)) {
3         return absvar(a,3);
4     } else {
5         return absvar(b,5);
6     }
7 }
```

Figure 2.9: A function with an example meta-mutant.

and the size of a test suite will generally increase in kind, the cost of mutation analysis can become prohibitively expensive, exceeding available computational resources. To counteract or mitigate this problem of *execution cost*, a variety of techniques have been proposed, which are described in this Section.

Due to the large number of techniques, a number of schemes have been proposed to classify the type of optimisation(s) they employ. Jefferson et al. [50] categorised techniques according to three types – "do fewer", "do smarter" and "do faster". These refer to reducing the number of mutants, reducing the cost per mutant or distributing their execution, and executing each mutant more quickly, respectively. Each of the techniques discussed in this Section are categorised according to this scheme.

Mutant schemata – *Do faster*

Due to the large cost that would be associated with compiling the entirety of each mutant program, where necessary for the language used, some mutation analysis systems made use of an interpreter-based approach, such as in the Mothra Fortran mutation system [66]. However, Untch [110] observed that using such an approach increased test execution time significantly, which generally represented the majority of time taken for mutation analysis. To address this issue, they proposed the *mutant schemata* approach, where each of the mutant programs are combined into a single program – the *metamutant* – that can be compiled once, but configured to execute a particular mutation via a parameter provided at runtime. This allows the test suite to be executed much more quickly, with the parameter being varied each time to run the program with a specific mutation applied. Figure 2.9 provides an example of a program and a meta-mutant using a similar style to the mutant schemata implementation of Untch [110]. In this case, two

different mutations are considered – replacement of relational operators, and replacement of variables with positive and negative absolutes values, implemented using the `relop` and `absvar` functions. The last operand in both functions identifies where the statement is in the program, which corresponds to particular entry in the configuration file. Each relational expression in the program (i.e., on line 2) has been replaced by a call to the function `relop`, which calls a particular relational operator on its operands according to the runtime configuration file. By configuring the behaviour like this, each mutation of the relational operator (i.e., $=,\neq,<,\leq,\geq$) can be "activated" at runtime, without a need to recompile the program. If a different mutation is being activated then the `relop` operation will simply apply the $>$ operator, as in the original program. Similarly, each variable reference in the program (i.e., lines 2, 3 and 5) is replaced with a call to `absvar`, which returns the variable directly when inactive, and either the positive or negative absolute value (e.g., $|\mathtt{a}|$ or $-|\mathtt{a}|$) when active, depending on which mutation is specified. It is noted that this can easily be combined with other schemata functions by nesting the calls, as shown on line 2. The mutant schemata approach has since been implemented into a wide range of mutation analysis systems [17, 33, 35, 84, 97, 111], with a selection of those that included an empirical evaluation discussed further below.

Untch et al. [111] performed an evaluation of the mutant schemata approach, comparing its efficiency with that of the interpreted Mothra mutation system. For this, a single C program was used which had to be transformed to a Fortran program for use with Mothra, producing 385 mutants and 364 for the C and Fortran variants, respectively. Their results showed that the schemata approach was 4.1 times faster, although given the greater number of mutants for the C variant the schemata approach is in fact more than 4.1 times quicker. The findings of this initial experiment, however, are clearly slightly limited given the use of only one case study program, as well as the inability to verify that the achieved mutation scores are comparable between both techniques, due to the differing number of mutants. In addition, differences in test suite execution times for the C and Fortran programs are not controlled for, nor discussed. However, when compared to interpreted mutation analysis systems, where the entire runtime of a particular language has to be precisely and correctly modelled for the mutation analysis results to be truly representative, the mutant schemata approach proposed is significantly easier to implement and less error prone, as it is compiled and executed using exactly the same tools as the original program.

The *Javalanche* tool [97] also makes use of mutant schemata to reduce the number of separate programs produced when mutating Java programs, in addition to making use of other optimisations (described elsewhere in this Section). In evaluating their tool, Schuler and Zeller showed that even for large programs – with up to 94,902 lines, 339 tests and producing 14,357 mutants – the execution time is not practically infeasible, taking less than 6 hours. Although this time may seem very high, it is still low enough to be plausible to execute in a practical setting on an occasional basis (e.g., daily) to provide a measure of test suite quality, even though it clearly cannot be used for near-interactive feedback for the tester. These results show that the optimisations such as mutant schemata can help to vastly improve the time taken for mutation analysis.

Madeyski [74] presented the *Judy* mutation analysis tool, which made use of the mutant scheamta technique to reduce the compilation overhead of mutation analysis, using Java and the AspectJ aspect-oriented programming library. This allows parts of the code to marked as "pointcuts" at which point fragments of code, each called an "advice", can be executed. Through this approach, each mutation can be modelled as an advice fragment that can be optionally executed, according to whichever mutation is desired. The performance of this approach was compared to that of the *MuJava* tool [89, 72, 73], which also uses the mutant schemata approach and other optimisations (discussed later in this Section). Comparing the two systems across 24 open-source programs from the Apache Software Foundation showed that Judy was approximately 12 times faster on average, processing a mean of 52 mutants per second. This demonstrates that some means of implementing the mutant schemata technique may be more efficient than others, and that in general the approach can help in the production of high performance mutation analysis systems that may help mutation analysis to be feasible in a practical setting.

Compiler integration – *Do smarter*

When mutating a program written in a language that must be compiled prior to execution, each extra mutant increases the time taken for compilation. While the mutant schemata approach (see page 58) aims to tackle this problem by producing a single large, meta-mutant program that contains each of the possible mutations, this still requires the original program to be parsed to produce the mutations and then the meta-mutant to be compiled. To reduce this potentially significant cost, DeMillo et al. [31] suggested applying a *compiler integrated* approach to mutation analysis, whereby the production

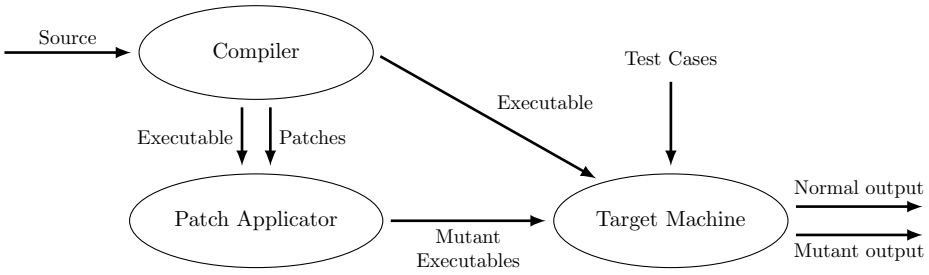


Figure 2.10: An overview of the compiler integrated approach of DeMillo et al. [31].

of mutants is implemented directly into the compiler rather than being a separate process. This reduces the amount of time spent parsing and compiling the original program, lowering the overall cost of mutation analysis.

In the approach of DeMillo et al. [31], a Fortran compiler was augmented to omit each mutant as a "program patch" – a compiled unit that could be applied at run-time to modify the program behaviour, according to the single mutation it contains. An overview of this implementation is shown in Figure 2.10. This avoids the cost of recompiling the code that would otherwise be duplicated across each mutant, whilst only adding a small overhead itself, although this specific implementation approach does rely on support for such runtime patching in the language in use. While the approach overall was described as being efficient, it has not been subject of an empirical evaluation, so it is unclear what degree of time saving it achieves.

The techniques of compiler integration and mutant schemata were combined by Just et al. [56], in an approach referred to as *conditional mutation*, implemented in a mutation analysis tool called *MAJOR* (**m**utation **a**nalysis in a **J**ava c**o**mpile**r**). Instead of using compiler patches, this approach, integrated as part of the standard `javac` Java compiler, automatically inserts conditional branches into the abstract syntax tree of the original program that contain the mutated versions of each modified statement. As with other mutant schemata techniques, these can be optionally enabled at runtime to adjust the program behaviour to that of a particular mutant. This approach was empirically evaluated by mutating eight open source programs with a total of 752,905 executable lines of code to produce 796,484 mutants [55]. Results of this initial step showed that the additional time taken for compilation due to the compiler modifications were negligible, while the increase in memory was still reasonable for execution on commodity hardware. Mutation analysis was then performed by executing the original test suite for each program,

revealing that the worst-case increase in time taken for executing the test suite varied between 1% and 30%. However, this is still likely to be a significant reduction compared to an approach without both optimisations, for which analysing 796,484 mutants would become infeasible. While these results show that conditional mutation, and so in turn compiler integration and mutant schemata techniques, can reduce the cost of mutation analysis it is unclear how easily it may be maintained over time as the target platform continues to be developed. In addition, it requires close integration into the tools used with a specific programming language, thus increasing the implementation cost of this technique for other languages. Furthermore, where the tools available for a specific language are not suitably extensible, or their source code freely available, such an approach may become prohibitively difficult to implement.

Bytecode mutation – *Do smarter*

Rather than compiling to machine code that can be directly executed on a given machine, a number of programming languages – such as Java, Erlang and Lua – are compiled to bytecode. This is a portable intermediate between source code and machine code that is later executed by an interpreter in a virtual machine. The *bytecode mutation* technique attempts to reduce the time taken for mutation analysis by operating on the compiled bytecode of a program, rather than the source, thus avoiding the need to compile each mutant separately.

Offutt, Ma, and Kwon [89, 72, 73] implemented the bytecode mutation approach, along with the mutant schemata technique, for Java programs in the *MuJava* tool (**Mu**tation System for **Java**). In this case, bytecode mutation is applied in a number of suitable mutation operators to produce each mutant as a separate class file. Each of these is then loaded at runtime using a Java class loader, where the chosen class determines which mutant is activated, and then tested with a series of unit tests using Java reflection. This approach was evaluated with 7 of 264 classes from the open source Byte-Code Engineering library, comparing the optimised approach against individual compilation of mutants – as usually required when mutating source code instead of bytecode. Results showed that on average mutants could be generated 9.3 times faster and executed 2.1 times faster, with an overall speed up of 5.5 times. Although the subjects of the experiment were limited, this still shows a significant potential for reducing the time needed for mutation analysis, improving its practical applicability.

Rather than implementing only a selected number of operators using bytecode mutation, as with the MuJava tool, Irvine et al. [47] implemented all of their operators using bytecode mutation in their Jumble tool. In this case, the operators replace each instruction in the bytecode with each other instruction that requires and produces the same number and type of arguments. For example, one mutant would be to replace a single instance of `dmul` – multiply two doubles to produce a double – with `dsub` – subtract two doubles to produce a double. This tool was used with programs with up to 310,000 lines of code, showing the technique does scale sufficiently, although an empirical analysis of the performance isn't provided. Nonetheless, anecdotal evidence did suggest that the mutation analysis process was useful in providing feedback to developers and testers of a real-world application in a industrial setting, where mutation analysis was otherwise believed to be prohibitively expensive, providing some support for bytecode mutation as an optimisation technique.

Incremental mutation – *Do fewer*

When using mutation analysis to evaluate and improve a test suite during the development of a program, the analysis process must be executed frequently to determine whether changes have altered the mutation adequacy of the test suite. However, given that many changes will only lead to small differences in behaviour, isolated to small parts of the program, the mutation adequacy will remain the same for the majority of the program. Therefore, a significant amount of the mutation analysis process is repeated between executions, executing the same tests against the same fragment of the program and mutants, producing the identical results. Cachia et al. [23] observed this inefficiency and proposed *incremental mutation testing* as a solution, which limits the scope of mutant generation to only those sections of code that have changed since the last mutation analysis execution, as well as only running the tests that will exercise those sections. Applying this process during the development of a program should reduce both the number of mutants examined and the amount of tests executed, decreasing the time taken for mutation analysis and improving its applicability in a practical setting. An implementation of this approach was evaluated with multiple versions of the Apache Commons CLI Library – an open source Java project with ∼5000 executable lines of code – comparing incremental and standard mutation analysis with three different degrees of "code churn" (how many lines were changed). Incremental mutation analysis was shown to reduce

the time taken for mutation analysis by approximately 90% in all three cases, removing between 45% and 91% of mutants. However, the mutation score decreased radically – between 0.94 and 0.45 for normal mutation and incremental mutation in the worst case – and with up to 3.4 times as many live mutants for incremental mutation. Cachia et al. suggested that this was likely a consequence of the naïve approach used for reducing the number of tests executed, suggesting that further work is required to improve the static analysis approach used to detect which tests to execute.

Higher order mutation – *Do fewer*

While the majority of mutation analysis techniques focus on the modelling of single small faults, as injected by applying one mutation operator per mutant, it may be that applying more than one mutation operator could produce more useful mutants. As first proposed by Jia and Harman [53], such *higher order mutants* (HOMs) may still represent simple programmer errors, yet be more valuable for evaluating test suites as they may be significantly harder to kill - therefore, killing a small number of HOMs could provide a better measure of fault-finding capability, without a significant increase in analysis cost. In particular, Jia and Harman identified a specific class of HOMs that were most likely to be useful, referred to as a strongly subsuming HOM (SSHOM). A subsuming HOM is harder to kill than the first order mutants (FOMs) it is constructed from - that is, there is a smaller range of inputs that would reveal the faults injected in the HOM. A SSHOM is one type of subsuming HOM, whereby any test that can kill it is also able to kill its constituent FOMs. Therefore, a SSHOM is able to replace these FOMs while still ensuring a mutation adequate test suite can detect at least the same range of faults.

Clearly, producing all possible HOMs would lead to a combinatorial explosion in the number of mutants. Instead, Jia and Harman [53, 51] investigated whether three different search algorithms – a greedy algorithm with random restarts, a genetic algorithm, and a hill climbing with random restarts – could automatically identify which HOMs were hardest to kill, and therefore were most likely to represent a subtle fault, according to the guidance of a fitness function. To evaluate the prevalence of different types of subsuming HOMs when mutating C programs, the higher order mutation approach was implemented in a tool called `Milu` [54]. This was executed with 10 C programs [51], which contained up to 6,000 executable lines of code and produced at most 68,843 FOMs. The results showed that SSHOMs can be successfully identified by this search process with over 8,000 being

found for the largest program, with the search using genetic algorithms being the most efficient. This shows that the number of SSHOMs for a program is likely to be significantly lower than the number of FOMs, thus giving good potential to reduce the time taken by mutation analysis. In addition, manual analysis confirmed that SSHOMs do indeed model subtle faults that require the tester to choose test inputs more carefully, and therefore represent good candidates to form a reduced pool of mutants for use in mutation analysis. A subsequent extension of this work including an additional 7 programs [42] gave further evidence to support the value of HOMs over first-order mutants, showing that a test data generation that aimed to kill HOMs was able to kill between 8 and 38% of mutants left alive by the next best approach from the literature.

Langdon et al. [69, 70] expanded upon this work, applying genetic algorithms, genetic programming and Monte Carlo sampling to search for hard to kill and realistic HOMs. In this case, the search process is a multi-objective problem guided by a fitness score that specifies HOMs must be syntactically similar to the original program – and thus, realistic as a programmer error according to the competent programmer hypothesis – as well as hard to kill. This confirmed that higher order mutation is able to produce mutants that model complex faults that FOMs cannot and that these are harder to kill than FOMS. In addition, this demonstrated that multi-objective search is able to find such FOMs in a reasonable time without specialised hardware.

Removing redundant mutants – *Do fewer*

While each mutation is intended to model a different type of fault, Just et al. [57] demonstrated that it is possible that some produced mutants are guaranteed to be killed by a test case that reveals another mutant. These are referred to as *redundant mutants*, which decrease the efficiency of mutation analysis while also reducing the accuracy of the mutation score – as it is unduly biased towards test suites that kill many of these mutants. For example, in a Java program given the statement `a && b`, the mutations `lhs` and `rhs` (return the left and right operand, respectively) return a different result from the original program when `(a,b) = (true,false)` and `(a,b) = (false,true)`, in turn, and would therefore be killed. This overlaps with the input required to detect the mutation `a || b`, which is killed for the same test inputs, and is therefore deemed redundant. Continuing this notion, Just et al. identified four mutations which were sufficient to test for the same behaviour as another six for statements with `&&` clauses, and

a further four to make another six redundant for || clauses. Adjustments were therefore proposed to the two operators responsible for these mutants, to prevent the production of these redundant mutants. An evaluation with four open-source programs, with a total of 71,041 executable lines of code, revealed that the affected mutation operators produced up to 57.8% of mutants, in the worst case. Therefore, as expected, applying the adjusted versions of the operators reduced the number of mutants significantly, with an average decrease of 24%. This in turn reduced the time taken for mutation analysis by between 9.9% and 34.1%, which is a wide, yet significant, range. Therefore, identifying and removing redundant mutants can yield meaningful improvements to the efficiency of mutation analysis, as well as positively impacting upon how accurate the mutation score is for evaluating a test suite.

Selective mutation – *Do fewer*

Originally proposed by Mathur [75], the *selective mutation* technique involves reducing the number of mutants used for mutation analysis by removing those produced by certain operators. Mathur suggested that this technique, initially referred to as *constrained mutation*, should have a linear complexity in the size of the program, as opposed to the quadratic complexity of full mutation, although may not be suitable for testing critical software. As such, it may be useful in cases where mutation using all available mutants has become too costly, due to the computational expense of compiling each – if necessary for the programming language being used – and executing, in the worst case, the full test suite.

***N*-selective mutation** Offutt et al. [87] experimentally evaluated selective mutation by implementing it according to the proposition by Mathur [75], for Fortran programs. The number of mutants was reduced by removing mutants created by the n most productive operators – those which produce the most mutants. This was referred to as n-selective mutation as opposed to *full mutation*, when mutants from all operators are used. To investigate whether selective mutation reduces the number of mutants from a quadratic complexity in the size of the program to a linear complexity, 28 Fortran programs were examined that ranged from 8 to 164 lines of code, producing a total of 81,159 mutants. These results showed that the number of mutants was still quadratic, as with full mutation, but that nonetheless 2-selective, 4-selective and 6-selective mutation can

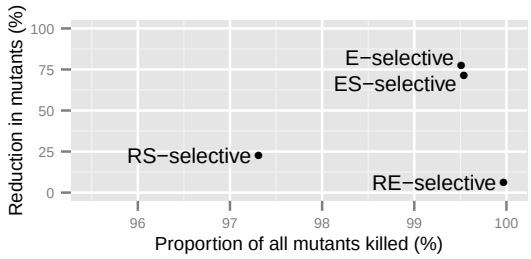
achieve a significant saving. In fact, further examination of these results showed that for these 28 programs the mutants created by the 1st and 2nd most productive operators represent over 30% of all mutants, the 3rd and 4th for over 20%, and the 5th and 6th for around 15%. Next, the selective mutation techniques were evaluated to see if they could still provide a tester with an accurate estimate of the quality of their test suite. This was measured by seeing whether a mutation adequate test suites for the selective sets of mutants was also mutation adequate for the full set of mutants. The Fortran programs used to test this include between 10 and 48 statements, producing 183 to 3010 mutants. All equivalent mutants were removed, and although an approach for this is not discussed it is assumed manual analysis was applied. For each of the three n-selective sets of mutants, *selective mutation adequate* test suites were then created, with test cases being generated automatically where possible or otherwise being hand written. These test suites were then used for full mutation analysis, executing them against all mutants for each of the programs. The test suite produced for 2-selective mutation was mutation adequate for the full set of mutants for 8 out of 10 programs, killed 99.94% of mutants in the worst case and 99.99% on average. This shows that the 2-selective mutants provide a very accurate prediction of whether a test suite will be mutation adequate for full mutation. It also suggests that those mutants produced by the two most productive operators are likely to be easy to kill. The reduction of mutants was also notable, with an average of $\sim$24% across all programs. Although the real-world time saving is not reported, it is clear that this likely represents a significant reduction in the computational cost. Results for 4-selective mutation and 6-selective mutation showed these techniques both provided good predictors of whether a test suite is adequate, with killing an average of 99.84% and 99.71% of mutants each, while both also greatly decreasing the number of mutants executed, reducing this by 41% and 61% respectively. While these results all suggest that selective mutation may greatly reduce the cost of mutation analysis, while still giving a tester a strong indication of whether their test suite is sufficient, there are a number of limitations. Firstly, it is unclear whether these results can be generalised to other programming languages besides Fortran, and whether they may be the product of specific characteristics of the mutants produced by the Mothra mutation system. Secondly, a low number of programs were studied and these were generally very small – only 10 programs with at most 48 executable statements, for the latter half of the experiment. In addition, the evaluation of the selective techniques only considers the case of selection adequate test suites, and does not examine whether sub-adequate test suites achieve similar mutation

scores for both the selective and full sets of mutant. Finally, the experimental design used requires both that all equivalent mutants be identified and test cases are produced to kill all other mutants – both of these represent potentially huge costs, making reproduction of this experiment with other languages very difficult. Nonetheless, these results do indicate that in some contexts particular mutants may be easier to kill, and therefore contribute less to the overall picture of whether a test suite is sufficient.

Randomly selected x% mutation Wong and Mathur [116] investigated how mutation with only two specific mutation operators, referred to as constrained mutation, and *randomly selected x% mutation*, where a random x% of mutants produced by each operator are used, compare to full mutation, where mutants from all operators are used. The two operators, *abs* and *ror*, were selected because of the type of test data the tester is required to produced in order to kill them. The *abs* operator creates mutants by surrounding the right hand-side of assignment expressions with positive and negative absolute operations (e.g., x becomes $|x|$ and $-|x|$), as well as `zpush` operation that can only be killed by setting the expression value to zero. These mutants force the tester to select values from varying parts of the input domain, including zero as well as positive and negative non-zero values. The *ror* operator produces mutants by replacing the operator in a relational expression with each other available operator (e.g., $x < 10$ becomes $x = 10$, $x \neq 10$, $x > 10$, $x \leq 10$ and $x \geq 10$) and either boolean constant (e.g., $x < 10$ become *true* and *false*). To kill these mutants, the tester must explore the possible boundary values in the predicates for each conditional branch of the program. This ensures that the specified constants in each relational predicate are correctly defined and that the tester specifies cases to explore all feasible branches. The approaches were evaluated by an empirical experiment using 4 partial Pascal programs, which were converted by hand into C and Fortran variants for compatibility with the various tools used. A total of 2,991 mutants were produced for these programs using the Mothra Fortran mutation tool. For randomly selected x% mutation, 7 levels were tested – 10 to 40% in 5% increments. The efficiency of each technique was measured according to the reduction in number of mutants, as well as the reduction in the number of tests required – which may represent another significant human-time cost. The effectiveness was measured using a similar metric to Offutt et al. [87], by measuring how close a selection adequate test suite was to being full mutation adequate. These test suites were produced by a combination of automated means and manual creation. All equivalent mutants were identified and removed – it is assumed, by

hand, as no other technique is mentioned. The results for constrained mutation showed that on average the selection adequate test suite was able to kill 96% to 99% of the full set of mutants, while reducing the number of mutants by 80%. Findings for randomly selected x% mutation showed that the test suites were able to kill an average of 95% of the full set of mutants, when x is between 10 and 20%, and 98%, when x is greater than 20%. Clearly, the reduction in the number of mutants is fixed for randomly selected x% mutation, varying between 90% and 40%, decreasing as x increases. These results suggest that removing the mutants produced by many mutation operators may well still provide a good indication of the quality of a given test suite, while significantly reducing the execution cost of mutation analysis. However, determining which operators yield these results may be specific to the programming language in use, as this may impact on what types of programmer mistakes are most common and also most important to detect. In addition, it is unclear whether these results apply only to selection adequate test suites – that is, it is not known whether a test suite that kills half of the selected mutants would also kill half of the full set of mutants. This may limit the practical value of this technique to a tester who may not be able to invest enough effort to kill all mutants, as it may give a poor indication of the quality of the test suite. If it provides an overestimate, the tester is given too much confidence in their test suite, while an underestimate may lead to extra unneeded human effort, to produce additional test cases. In addition, the experiment only considered extracts from a small number of programs, raising concerns of generalisability of the results. Expanding the empirical study using the same design, however, would require significant effort to both manually identify equivalent mutants and produce mutation adequate test suites.

Categorising by operator types Offutt et al. [88] observed that the mutation operators used in the Mothra Fortran mutation system could be divided into three groups, according to the type of element they mutate, and then used these to produce sets of operators to use in a selective mutation experiment. Firstly, the *Replacement-of-operand* operators substitute each operand in statements with each other valid operand. Secondly, the *Expression modification* operators replace existing operators and add new operators. Finally, the *Statement modification* operators replace entire statements. These were formed into four sets, named according to the operators they included – ES-selective (expression and statement modification), RS-selective (replacement and statement modification), RE-selective (replacement and expression modification) and E-selective (only



Operator set	Mutants killed (%)	Reduction in mutants (%)
ES-selective	99.54	71.52
RS-selective	97.31	22.44
RE-selective	99.97	6.04
E-selective	99.51	77.56

Figure 2.11: The proportion of all mutants killed by test suites produced with different subsets of mutation operators and the reduction in number of mutants analysed, as reported by Offutt et al. [88].

expression modification). Each of these selective sets were evaluated using 10 Fortran programs of between 10 and 48 statements, for which 183 to 3,010 mutants were produced with the full operator set. As with prior works [116, 87], the effectiveness of the selective techniques were assessed by forming a selection adequate test suite for each and evaluating what proportion of mutants it could kill from the full set of mutants. This again called for the creation of adequate test suites, which required a combination of automatic data generation and manual test case creation for hard to kill mutants, and the manual identification of equivalent mutants, which would otherwise reduce the mutation scores to below 1. The results of this are shown in Figure 2.11, with both axes of the plot representing "higher is better" metrics. From this, the E-selective (replacement and addition of operators) operator set appears to be the most compelling choice, with the greatest reduction in mutants and an approximately equal mutant killing ability to the ES-selective operator set. Given the similarity to prior evaluations of selective mutation, many of the limitations are shared – for example, the necessity to produce mutation adequate test suites (for some given set of mutants) and to remove all equivalent mutants, both of which may prove incredibly time consuming for all but small programs. In addition, these results may be highly dependent on the specific set of operators being used for a given programming language – in this case, Fortran – as these results may be an artefact of an overlap or subsumption relationship between mutation operators, where a test that kills mutants produced by one operator are guaranteed to kill those produced by another.

Cost-based heuristics Mresa and Bottaci [79] analysed each of the Mothra Fortran mutation operators separately, to find what their individual mutation scores and execution costs were. This was then used as a heuristic to guide the choice of operators for selective mutation. The cost for each operator is considered to be not only the number of mutants it produces, as with much of the prior literature, but also the proportion of equivalent mutants generated as well as an estimate of the cost of generating test cases. This highlighted the need for test cases to be generated such that tests are only included in the test suite if they kill a mutant that is not killed by at least one other test – otherwise, they are *redundant*, in that they do not contribute fault-finding capability to the test suite, and represent an unnecessary extra execution cost. An experimental evaluation of the operator mutation scores and execution costs used 11 Fortran programs, with an average of ∼44 lines that led to a production of 35,321 mutants using 21 of the Mothra mutation operators. For this, selection adequate test suites were created for each operator by a combination of automated means, using a constraint-based tool and/or random data generation, and hand written tests, for hard to kill mutants. Mutants for which no automatically generated or manually produced test case could be created were deemed equivalent and removed. As with prior studies, applying the selection adequate test suites to the full set of mutants in the first half of the experiment revealed that these tests were still able to kill over 70% of mutants, in the worst case, and over 98%, in the best case. The results also showed that the cost associated with the operators positively correlates with the mutation score, suggesting that the larger size of the produced test suite may contribute to this increase in mutation score. The latter half of the experiment looked into how these operators could be most effectively be grouped into sets for use in selective mutation showed that the best choice of operator selection technique depends upon the needs of the tester. If the tester wishes to achieve a mutation adequate test suite, then randomly selected $x\%$ mutation as applied by Wong and Mathur [116] is likely to be more efficient, when considering the extra costs of equivalent mutant identification and test case generation. On the other hand, if the tester relaxes this requirement, then a more efficient subset of the available mutation operators may be used to significantly reduce the cost of mutation analysis. While Mresa and Bottaci provided a much more comprehensive measurement of the cost of individual mutation operators than prior work, it is slightly unclear how well the cost values relate to the actual human and computational time costs associated with them. In particular, the weightings used for individual components of the measurement may require additional investigation to determine their

degree of correctness, for example by measuring the actual time taken in a controlled human study. Nonetheless, the results do provide additional understanding of the high degree of overlap between mutants produced by the Mothra mutation operators, such that there may well exist subsumption relationships between them.

Criteria for operator selection Barbosa et al. [10] applied some of the knowledge of selective mutation for Fortran programs presented Offutt et al. [88] instead to mutation of programs using the C programming language. This led to the definition of a generalised guideline for selecting a subset of mutation operators from any given full set of operators. This involves choosing operators according to six criteria, including using operators that lead to the production of test suites with the highest full-set mutation score, using at least one operator from each group or class, and evaluated where mutants produced by different operators may overlap. This guideline was experimentally evaluated by implementing it as a selection procedure, which was applied to the mutation analysis of 27 small C programs ranging from 11 to 71 statements. The full set of 71 mutation operators from the Proteum mutation tool were applied to these programs, although only 39 generated at least one mutant, producing a total of 20,146 mutants. A second phase of the experiment introduced 5 alternative C programs of 76 to 119 lines, which led to the production of a further 12,834 mutants. Applying the procedure to these operators and mutants yielded a set of ten mutation operators, which achieved a mutation score of 0.997 while using only 35% of the original number of mutants. This was compared to prior techniques such as 6-selective mutation [87] and randomly selected x% mutation, and was shown to produce the highest mutation score. The randomly selected 40% mutation technique was shown to achieve a marginally lower mutation score whilst slightly improving upon efficiency, however it is notable that random selection may suffer from unreliable results from its inherent stochasticity. Overall, these results showed that the suggested guideline was able to assist in the identification of an effective subset of mutation operators for the C programming language from those provided by the Proteum mutation system. These operators provided a more accurate model of the mutation score obtained with the full set of operators than other techniques previously suggested for selective mutation. However, it is unclear how the results may have been affected if a larger proportion of the Proteum mutation operators had been applicable (i.e., generated at least one mutant), as only 55% and 79% of operators were considered in the first and second phase of the experiment, respectively.

Statistical procedures Namin et al. [81] proposed treating the problem of selecting mutation operators as a statistical variable reduction, or feature selection, problem. Variable reduction is a process used to simplify a model of some problem, by reducing the number of predictor variables, whilst still ensuring it is accurate. In the context of selective mutation, the model is aiming to predict the mutation score a given test suite would achieve for full mutation, based upon the proportion of mutants killed for each included mutation operator. Therefore, the mutants produced by each operator represent a variable that may or may not be included in the model, according to a variable reduction technique. The subset of operators is produced using a cost-based least-angle regression (CBLARS) procedure that is able to find subsets of predictor variables while avoiding the need for producing all possible subsets, which the authors claim would require a clearly infeasible 2^{38} models. As well as optimising the accuracy of the selective mutation score as a predictor of the full-set mutation score, the cost-based element of the procedure reduces the number of mutants included, as opposed to the number of operators included as in a standard least-angle regression procedure. The approach was evaluated using seven programs written in C with a total of 2,188 executable statements. As these programs, known as the Siemans programs, have been studied multiple times previously they are accompanied by a large number of test cases. Sampling a wide range of test cases from this pool provides the CBLARS procedure with a wide range of mutation score values, ensuring the results are representative for all scores – not just mutation adequate scores, as with previous work. Applying the 108 Proteum operators to these programs produced 70,238 mutants, which were sampled to give 2,000 mutants per program, divided proportionally for each mutation operator. Applying the CBLARS procedure to these mutants identified a set of 28 mutation operators that are able to reliably predict the full-set mutation score (with an adjusted R^2 value of 0.98, indicating a high goodness of fit), while reducing the number of mutants produced by 92%. In practical usage, this allows a tester to apply those 28 mutation operators (provided the scores achieved for mutants by each is recorded), and use the coefficient values reported by Namin et al. [81] to calculate a reliable estimate of the full mutation score.

Parallel execution – *Do faster*

Mutation analysis requires many mutants to be produced, compiled and executed via a potentially large number of test cases. With all but small programs, this can become

very expensive to execute. However, while there may be a specific order tests must be executed in, there are not dependencies between the execution of each mutant - clearly, if the execution of one mutant could impact the execution of another, it would not be possible to determine which was responsible for any change in behaviour, which may become unpredictable at best. It is therefore possible to parallelise the mutation analysis process, executing the tests against multiple mutants simultaneously by exploiting suitable hardware resources, such as multi-core processors and compute clusters.

Early approaches using parallel techniques for mutation analysis focussed on exploiting specialised parallel hardware. For example, Byoungju and Mathur [22] developed a parallelised version of the Mothra mutation system, called P^{M}othra, which allows the tester to transparently utilise a range of parallel hardware, including an Ncube hypercude computer and a vector multiprocessor. This was evaluated with five programs with a total of 138 lines, producing 10,391 mutants, and 150 test cases. Overall, this showed a speed-up for mutation analysis of between 40 and 50 times. These results demonstrated that given sufficient hardware capability it is possible to improve the time needed for mutation analysis, even if the hardware cost limit the applicability in an industrial setting rather than an academic one.

More recently, Schuler and Zeller [97] have made use of parallel execution in their Javalanche tool, which implements mutation analysis for Java programs. In this case, the multi-processing capabilities of commodity hardware can be used, in addition to distributed execution using several separate computers. As discussed previously (see page 59) this tool incorporates numerous other optimisation techniques, such as mutant schemata and selective mutation. As a result, Javalanche is able reduce the time taken for mutation analysis, making its use feasible for projects much larger than otherwise possible – for example, taking less than 6 hours to process all mutants for 14,357 mutants of a 94,902 line program.

Dynamic analysis – *Do fewer*

While techniques such as mutant schemata and selective mutation aim to reduce the cost of mutation analysis by using information about the mutants prior to execution, dynamic analysis takes advantage of information only available during runtime. Just et al. [58, 59] made use of such a technique in an attempt to reduce the number of tests

executed for each mutant, by identifying those tests who would not be able to detect a particular mutation. This is achieved by executing the test suite once against an instrumented the original program, recording which mutations could be detected by each test. In the simplest case, this involves identifying those tests that will not execute the mutated statement. Next, this analysis checks for each test and each mutant whether the mutation is able to change the state of the program – referred to as an *infected execution state*. Following this, the tests are checked to ensure that the mutated state would propagate and affect the overall behaviour of the section of code it is referenced in. Finally, where the state would be infected by a test, the infected execution state is compared to the state each other test would produce, including each unique state only once. Because each of these optimisations can be applied from a single execution of the test suite, they have the potential to greatly reduce the number of test case executions, and therefore the overall cost of mutation analysis. To evaluate the technique, it was applied to mutation analysis of 14 open source Java programs with a total of 669,100 lines of executable code, with test suites used being a combination of those developed alongside the programs themselves and automatically generated with the *EvoSuite* tool [34]. Overall, this empirical experiment revealed the three optimisations were able to reduce the time taken for mutation analysis by an average of 40%. This suggests that analysing the effect each mutant has on state by executing the tests once and reducing the number of tests by exploiting this information may prove a cost effective means of improving the efficiency of mutation analysis in other domains.

2.3.6 Mutation analysis with databases

While limited in quantity, there are a number of items of literature that apply mutation analysis in the context of relational databases, some of which have already been discussed earlier in this Chapter. This Section describes the most prominent of these, which form the most related literature to the content of this thesis.

As discussed in Section 2.2.2, Tuya et al. [106, 107] proposed a range of mutation operators to perform syntactic changes to SQL `SELECT` statements, implemented in the *SQLMutation* tool. As these types of query are used whenever retrieving data from a database, there is significant scope for a mistake in such a statement to cause a critical failure in a database application, therefore it is important to model such faults. A wide

range of faults were modelled including those relating to joins, ordering and predicates, as well as logical operators and the application of `null` values. An example of how these operators could be applied to a `SELECT` statement has been shown previously in Figure 2.5 (page 37). These mutants were used to evaluate the fault-finding capability of the data provided with the NIST SQL conformance test suite, which was able to detect 70% of the injected faults, subsequently improved to 85% and then 100% by applying automatic data generation techniques and incorporating manually crafted test cases, respectively. This suggests that the types of faults modelled can be difficult to detect, suggesting the impact they have on the behaviour of the query is subtle, but also importantly that these mutants are not equivalent as tests can be created that detect them, confirming that they could represent a real-world fault in an application that might lead to a failure. When compared to the contributions of this thesis, the work of Tuya et al. is instead focussed on the mutation of a different element of a database application. The mutation operators proposed in Chapter 4 modify the constraints in the `CREATE TABLE` statement used to describe the structure of the database, altering the data considered valid by the DBMS. In contrast, the operators defined by Tuya et al. make changes to the SQL `SELECT` statements used to extract rows of data from an existing database. These are therefore distinct areas of research, although it is possible they may be used to compliment each other as part of a collection of techniques to fully test a database application.

Extending upon the SQLMutation tool, Zhou and Frankl [120] developed the "*Java Database Application Mutation Analyser*" (usually shortened to *JDAMA*). When applied to a database application, this tool identifies where the Java database connectivity (JDBC) library is used to communicate with a database and instruments the underlying Java bytecode to allow mutation analysis of the SQL statement to be performed. This mechanism enables the mutation to be performed in the wider context of the application during testing, allowing values of program variables in a `SELECT` statement to be considered. This in turn allows the response from the database to be compared for the original `SELECT` statement and a series of mutant statements, using the same program state for each. For example, given the following Java statement that could form part of a database application:

```
String statement = "SELECT * FROM stock WHERE quantity < " + min + ";"
```

In this case, the `SELECT` statement is being formed through string concatenation[3], with the value of the variable `min` being substituted into the statement at runtime. To test mutants of this statement it is therefore important to consider the possible values of `min`. By capturing each `SELECT` statement as it is submitted to the database, JDAMA is able to execute each relevant mutant statement (generated using the SQLMutation tool) and compare the result against that of the original statement, to determine if the mutant is killed. As with the SQLMutation tool it is built upon, the JDAMA tool differs from the work contained in this thesis in that it is designed to operate on `SELECT` statements rather than the `CREATE TABLE` mutants. It does however highlight a possible path to integrating the work of this thesis into the wider context of testing database applications, using automatic instrumentation of JDBC statements to dynamically identify where mutations can be applied during application testing. However, as `CREATE TABLE` statements are usually static in nature, and as such would not include concatenation of program variables, the production of abstract statements that can later be transformed into concrete statements during execution would not be necessary in this context.

To generate data for use in database application testing, it is necessary consider both the constraints within the application and expressed as part of the SQL `SELECT` statements used to extract data from the database, declared in the `WHERE` part of the statement. Implemented as a framework used in the *MutaGen* tool, Pan et al. [92] proposed a technique that extracts the constraints in the `WHERE` clause and encodes them as normal application code, such that they can be analysed using an existing symbolic execution approach. This can then be used to infer constraints for both the program and database during execution, by capturing database interactions using an instrumented synthetic database. The MutaGen tool leverages this framework to generate data targeted to kill two different sets of mutants – mutants in the original program code, and mutants in embedded SQL queries. An evaluation of this approach showed data generated by the tool was able to kill significantly more mutants of both considered varieties than prior tools, although the evaluation included only two case study applications. Similarly to the other items of literature described in this Section, the work of Pan et al. is distinct from the contributions of this thesis in that it focussed on the `SELECT` statements executed

[3]The `SELECT` statement could otherwise be formed using a `PreparedStatement` which similarly substitutes variables into the resulting SQL, although automatically handles specific cases of character escaping to reduce possible security vulnerabilities. Although JDAMA does not explicitly support these statements, its authors note similar instrumentation could be developed to add support for them.

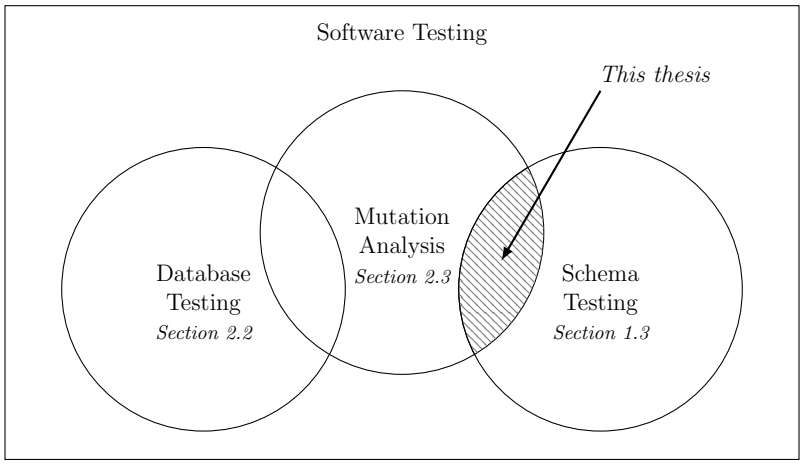


Figure 2.12: An overview of how the contributions of this thesis relate to the existing work discussed in this literature review.

within a database application, rather than the `CREATE TABLE` statements that define the structure and constraints of the database itself.

2.4 Summary

Measuring the quality of a test suite is important to both testers – who wish to be given assurance their program contains few, if any, faults – and researchers – who want to evaluate the tests produced by different techniques and compare their effectiveness. Test quality metrics such as coverage criteria (Section 2.1.2) can quantify how much the application has been tested, according to different notions of how thorough tests should be. Alternatively, mutation analysis (Section 2.3) can be used to estimate the fault-finding capability of a test suite, by measuring how many artificially seeded faults it can identify. Through the definition of mutation operators, these injected faults can model a wide range of real-world faults, thus providing the tester a good metric with which to evaluate the quality of their test suite.

While mutation analysis has been applied to a wide range of different domains, including database application testing as discussed in Section 2.3.6, it has yet to be used for measuring the quality of test suites that exercise the integrity constraints of relational database schemas. As discussed in Section 1.3, mistakes in the constraints of a schema

may lead to critical failures in programs that use it to make a database, leading to either a degradation of data quality or potential data loss – both costly problems for businesses to resolve. However, to date, no known work has been undertaken to evaluate the quality of test suites produced to test integrity constraints. This thesis investigates the application of mutation analysis to such test suites to provide a metric that can be used to compare the quality of suites produced by different techniques. In addition, the resulting mutation scores may be useful to testers to determine whether their test suite is sufficient to reveal misspecified schema constraints.

The next Chapter describes the tool in which the mutation analysis framework has been implemented. The subsequent Chapters then define how the integrity constraints of a schema can be mutated (Chapter 4), use the mutation analysis framework to compare a series of data generation criteria (Chapter 5), describe algorithms to automatically identify types of unwanted mutants such as equivalent mutants and evaluate their implementation (Chapter 6), and detail how various optimisations inspired by those in Section 2.3.5 can be applied to this new domain (Chapter 7).

Chapter 3

The SchemaAnalyst tool

3.1 Introduction

Having outlined how the work in this thesis relates to prior literature on database testing and mutation analysis in the prior Chapter, this Chapter now describes various important parts of the SchemaAnalyst tool, which is used as the basic foundation for the implementation of the techniques proposed in this thesis. The majority of the tool's functionality relies on an intermediate representation of SQL schemas, whereby the various tables, columns and constraints are represented as a graph of Java objects. These objects can then be inspected and manipulated programmatically, for example to detect the presence of particular patterns of interest or to make small modifications to produce mutant versions of the schema. The SchemaAnalyst tool includes classes that use the parse tree created by an SQL parser[1] to generate a Java source file from an SQL schema, which produces this object representation at runtime. This SQL parsing functionality has been used to produce intermediate representation versions of a variety of schemas from both real-world and synthetic sources. SchemaAnalyst also includes a data generation component that facilitates the production of test data, which can be used to exercise the constraints in a schema. An SQL writing and DBMS interaction component allows the schemas and data in the intermediate object representation to be transformed into SQL

[1]The SQL is parsed using a commercially available SQL parser, General SQL Parser (`http://www.sqlparser.com/`)

and executed with a number of different DBMSs. Finally, a framework is provided for producing mutated copies of SQL artefacts.

This Chapter aims to provide the following:

1. An explanation of the intermediate object representation of SQL used by the SchemaAnalyst tool to represent relational database schemas;
2. A description of the parsed SQL schemas available within SchemaAnalyst, including where their source and detailing of their attributes, which are used later in empirical experiments in this thesis;
3. Discussion of the range of data generation algorithms and coverage criteria implemented in the SchemaAnalyst tool to generate test suites for relational database schemas; and
4. A brief overview of the mutation framework implemented in SchemaAnalyst, which can be used to define operators that generate mutants of SQL artefacts.

3.2 Architecture

3.2.1 Overview

In Chapters 4 and 6, the algorithms given are described in terms of their implementation within the SchemaAnalyst tool, exploiting the SchemaAnalyst representation of SQL schemas. To provide a thorough understanding of the structure and functioning of this tool prior to its usage in these Chapters, this Section now describes its architecture and implementation.

Figure 3.1 provides an overview of the architecture of the tool, showing how its components can be used to perform mutation analysis of database schemas. The "Mutation" section of this Figure represents original work produced during my PhD. The SQL schema is transformed into the SchemaAnalyst intermediate representation of SQL using a third-party parser, GeneralSQL Parser, and the SQL Parser component.

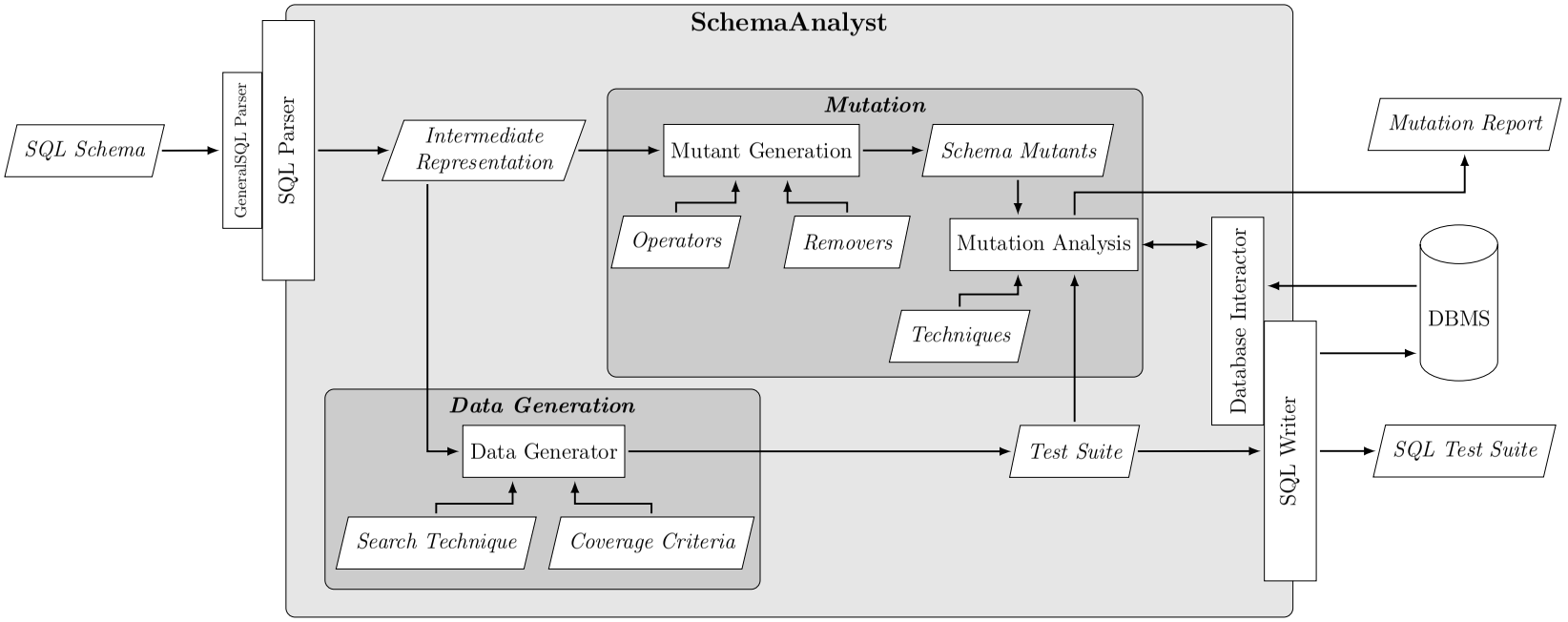


Figure 3.1: An architectural overview of the major components in the SchemaAnalyst tool.

Rectangular boxes represent processes or components of the tool, while slanted rectangular boxes depict inputs and outputs for these processes. Arrows show where data flows between processes and the direction of this. The light shaded area shows the boundaries of the SchemaAnalyst tool itself, while the darker shaded areas show the scope of two major components – *Mutation* and *Data Generation*, the former which is a novel contribution of this thesis in its entirety.

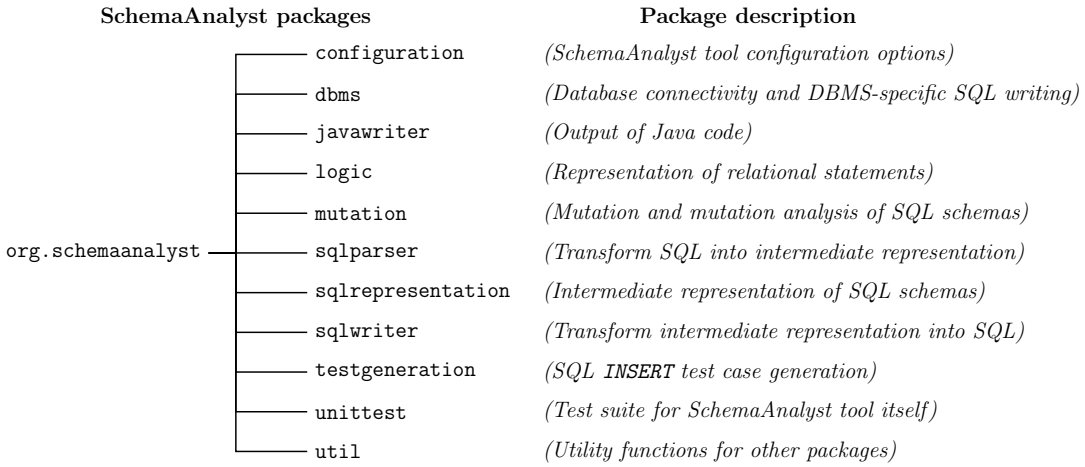


Figure 3.2: The main packages of SchemaAnalyst with descriptions of their purpose.

Once transformed into the Java object-based representation the schema can be used to both produce schema mutants, using the Mutation components, and produce test cases, using the Data Generation component. Mutation analysis can then be performed, according to some analysis technique, executing the test suite via a DBMS-specific Database interactor and SQL Writer. Finally, a mutation report is produced that contains the results of the mutation analysis process. This Chapter now describes parts of the SchemaAnalyst tool in more detail, providing the necessary background for the later Chapters of this thesis.

3.2.2 Components and package structure

The Java classes implementing the various functions in the SchemaAnalyst tool are divided into a number of different Java packages, the top-level of which are listed in Figure 3.2. These can be organised into three major components, which each expose some externally accessible functionality:

- *Input/Output*: `javawriter`, `sqlparser`, `sqlwriter`, `dbms`.
- *Data Generation*: `testgeneration`.
- *Mutation*: `mutation`.

The remaining packages are either only used internally within the SchemaAnalyst tool itself (`configuration`, `logic`, `sqlrepresentation`, `util`) or for testing purposes (`unittest`). The *Input/Output* component facilitates the transformation of relational database schemas expressed in SQL to and from the SchemaAnalyst intermediate representation (Section 3.3), including support for three different DBMSs (Section 3.3.3), as well as producing Java source code files from the intermediate representation. In addition, it allows other components of SchemaAnalyst to communicate with a DBMS, by handling creation of `INSERT` statements from generated data and wrapping functionality provided by the Java Database Connectivity (JDBC) API[2]. The *Data Generation* component (Section 3.5) uses search-based techniques to produce data that can exercise the constraints of a relational database schema, according to some coverage criterion (evaluated in Chapter 5). This data can be transformed into SQL using the Input/Output component to produce a test suite of `INSERT` statements. Finally, the *Mutation* component enables mutation analysis experiments to be performed, to evaluate how effective a given test suite is at detecting possible faults in a relational database schema. This includes a mutation framework (Section 3.6) that I have used to express a set of 14 mutation operators (Chapter 4), which each model a different possible fault in the schema. It also contains functionality I have implemented for automatically identifying three types of unwanted, or "ineffective", mutants (Chapter 6) and a collection of optimised techniques for reducing the computational cost of mutation analysis (Chapter 7).

3.3 Intermediate Representation of SQL

The `sqlrepresentation` package in the *Input/Output* component of SchemaAnalyst provides a collection of Java classes that can be used to represent an SQL schema as a graph of objects. These objects can then be manipulated programmatically (i.e., to produce schema mutants) and transformed back into SQL using the *SQL Writer* component of SchemaAnalyst. Although there may be minor differences in the SQL used by different DBMSs (sometimes referred to as *dialects* of SQL), this representation is DBMS-agnostic to allow for support of schemas from a number of different DBMSs (currently support for PostgreSQL, HyperSQL and SQLite is implemented). Differences between DBMSs

[2]JDBC enables Java programs to communicate with numerous SQL databases via a programmatic interface, although this requires all database statements to first be transformed into SQL statements.

is instead handled in the *SQL Parser* and *SQL Writer* components of SchemaAnalyst. A UML diagram showing the hierarchy of these classes is shown in Figure 3.3, although all methods and some attributes are omitted for clarity. This shows that while an SQL schema appears to be a relatively simple structure – consisting of tables, columns and constraints – these can mask the inherent complexity of SQL, such as the wide range of different data types and the highly flexible nature of `CHECK` constraint expressions. This Section now describes the classes of this intermediate representation in more detail, including how these can be used to model an example SQL schema.

3.3.1 Overview of classes

In the intermediate representation a `Schema` object acts as the root of the graph of objects, containing a variable number of tables and constraints. Each SQL table is stored as a `Table` object, which in turn references a collection of `Column` objects that each represent one column in the schema. A column is specified by its name and `DataType` (such as `Int`, `Varchar` etc.), where each `DataType` maps to one or more SQL data types according to SQL dialect being parsed. Each constraint is modelled as a specific implementation of the `Constraint` class, which consists of a reference to a `Table` and an optional name. The classes representing `PRIMARY KEY`, `UNIQUE` and `FOREIGN KEY` constraints each contain a list of affected columns, with the latter also including the referenced table and columns as `refTable` and `refColumns`, respectively. As a `NOT NULL` constraint only applies to a single column, the `NotNullConstraint` class contains a single `Column` attribute.

While the other types of constraints are limited in their complexity, a `CHECK` constraint contains an arbitrarily complex expression that may include many sub-expressions, which themselves may consist of multiple expression that each need to be stored. To allow the programmatic traversal and manipulation of these boolean expressions, the SchemaAnalyst intermediate representation includes a large number of classes to specify the expression of a `CheckConstraint` instance, using a tree structure. The classes can therefore be divided by whether they represent "leaf" expressions, which either refer to a column or some constant value, or "tree" expressions, which use references to other expressions to form some more complex structure. An example of the latter is a relational expression such as `price < 100` that would be represented using the `RelationalExpression` class with a `lhs` (left-hand side) attribute of a `ColumnExpression` referring to `price`,

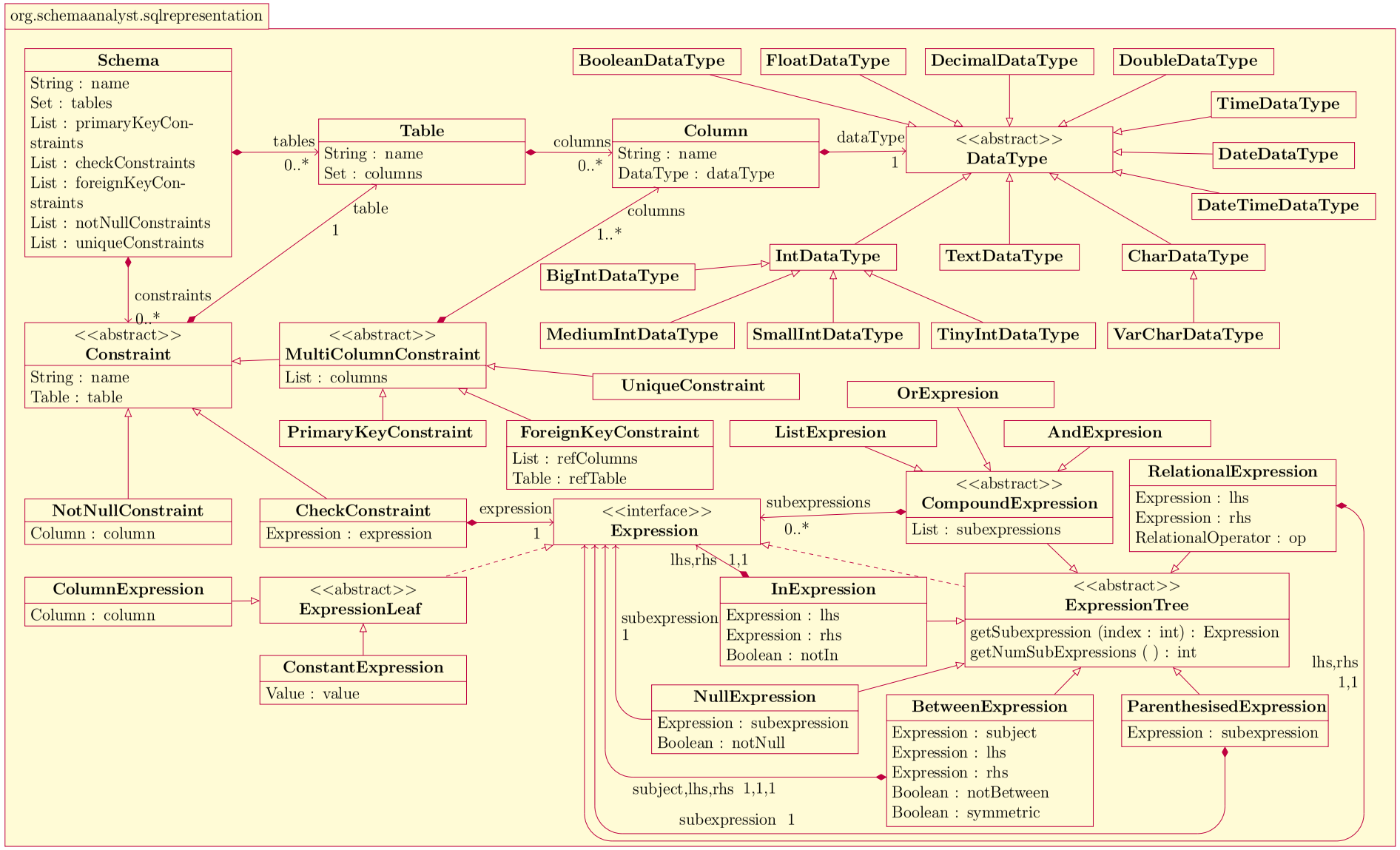


Figure 3.3: A UML diagram of the intermediate representation of SQL in the SchemaAnalyst tool.

a `rhs` (right-hand side) attribute of a `ConstantExpression` with the value `100`, and the less-than operator as the `RelationalOperator` attribute `op`. By compounding those classes shown in Figure 3.3 implementing the `Expression` interface, it is possible to represent a significant portion of `CHECK` expressions, including all of those for the schemas described later in Section 3.4.

3.3.2 Schema example

Figure 3.4 demonstrates how an example SQL schema is stored in the SchemaAnalyst intermediate representation, in terms of both the Java object graph and Java source file (as produced by the `javawriter` package), showing how different types of constraints are represented using the classes discussed in Section 3.3.1. While the Java source expression of the SchemaAnalyst intermediate representation is usually more verbose than SQL, it is worth noting that this is generally produced automatically using the SQL parsing and Java writing capabilities of the *Input/Output* component of SchemaAnalyst and is not usually interacted with by the user — instead, the Java source code is produced automatically by this component, and the resulting structure is manipulated programmatically at runtime.

3.3.3 Supported DBMSs

As the SchemaAnalyst intermediate representation of SQL is DBMS-agnostic, support for different DBMSs is implemented in the *Input/Output* component of SchemaAnalyst, using classes in the `dbms` package. Where necessary these encompass the small differences between DBMSs, such as formatting of identifiers and substituting equivalent data types where the exact data type is not available. Currently, SchemaAnalyst supports three DBMSs — PostgreSQL, HyperSQL and SQLite– that were chosen due to their differing design principles and architectures.

PostgreSQL [95] is a standalone DBMS used in a client-server configuration, which supports a large portion of the SQL specification, meaning a wide range of SQL features can be used. It is designed to be highly scalable in both table size and number of concurrent user connections, supporting tables of up to 32 terabytes and claiming a theoretically

Figure 3.4: The *ArtistSimilarity* schema, used to store musical artists that are similar to one another; as expressed in SQL, the SchemaAnalyst Java representation, and a diagram of the objects as instantiated by executing the Java code.

SQL

```
CREATE TABLE artists (
  "artist_id" text PRIMARY KEY
);

CREATE TABLE similarity (
  "target" text,
  "similar" text,
  FOREIGN KEY(target)
    REFERENCES artists("artist_id"),
  FOREIGN KEY("similar")
    REFERENCES artists("artist_id")
);
```

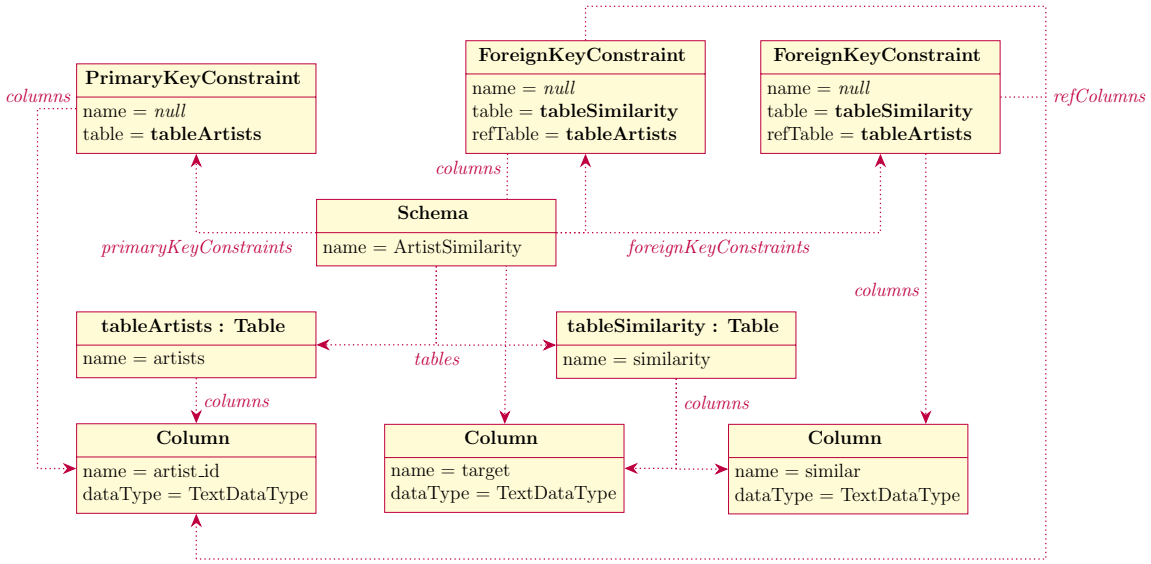
SchemaAnalyst representation

```
public class ArtistSimilarity extends Schema {
  public ArtistSimilarity() {
    super("ArtistSimilarity");

    Table tableArtists = this.createTable("artists");
    tableArtists.createColumn("artist_id", new TextDataType());
    this.createPrimaryKeyConstraint(tableArtists, tableArtists.
        getColumn("artist_id"));

    Table tableSimilarity = this.createTable("similarity");
    tableSimilarity.createColumn("target", new TextDataType());
    tableSimilarity.createColumn("similar", new TextDataType());
    this.createForeignKeyConstraint(tableSimilarity,
        tableSimilarity.getColumn("target"), tableArtists,
        tableArtists.getColumn("artist_id"));
    this.createForeignKeyConstraint(tableSimilarity,
        tableSimilarity.getColumn("similar"), tableArtists,
        tableArtists.getColumn("artist_id"));
  }
}
```

Instantiated object diagram



unlimited maximum database size, although results later in this thesis suggest it incurs some computational overhead for these advanced features.

HyperSQL [105] is an embedded database, executed as a component of a wider application, that supports a large number of SQL features whilst providing quick performance by reducing the overheads of the traditional client-server model. Applications using HyperSQL include the OpenOffice software suite and the JBoss Java application server, both of which have very large numbers of users.

SQLite [103] is also an embedded database that produces portable, single-file databases, which supports a reduced number of SQL features compared to HyperSQL and PostgreSQL. This enables for a very small overhead, leading to very wide adoption in applications such as the popular Internet browsers Mozilla Firefox and Google Chrome, as well as the Android mobile device operating system [102].

3.4 Database Schemas

Utilising the SQL parsing component of SchemaAnalyst, a collection of example relational database schemas expressed in the internal representation of SQL have been produced, by myself and the other developers of SchemaAnalyst. These were chosen to be representative of a wide range of schemas by selecting examples from a wide range of sources such as open source software projects, DBMS tutorial samples and textbooks. In addition, these vary substantially in the number of tables, columns and constraints they contain, and include examples of all of the major types of constraint supported in SchemaAnalyst. Table 3.1 provides a description of the source for each schema, as well as whether it is real-world or synthetic, and any references of publications that have previously included that schema. Table 3.2 lists these attributes for each of the schemas currently included within SchemaAnalyst.

3.5 Test Case Generation

To facilitate the testing of relational database schemas SchemaAnalyst contains a *Data Generation* component, which produces test data to exercise the constraints specified

Table 3.1: Description of parsed schemas in SchemaAnalyst

Schema	Type	Source	References
AdmissionsPatient	Synthetic	Microsoft Access example	
ArtistSimilarity	Synthetic	MillionSong research data set of music meta-data	[12]
ArtistTerm	Synthetic	MillionSong research data set of music meta-data	[12]
BankAccount	Synthetic	Teaching material	
BioSQL	Real-world	Storage of biological sequences and annotations	
BookTown	Synthetic	PostgreSQL example	
BrowserCookies	Synthetic	Worked example inspired by Mozilla schemas	
ChromeDB	Real-world	Google Chrome	
Cloc	Real-world	'CLOC' code line counter program export format	
CoffeeOrders	Synthetic	Textbook example	
Crafts2002	Synthetic	Textbook example	
CustomerOrder	Synthetic	Textbook example	
DavilaDjango	Real-world	Schema visualization tool	
DHDBookstore	Synthetic	Textbook example	
DellStore	Synthetic	PostgreSQL example	
Employee	Synthetic	Textbook example	
Examination	Synthetic	Textbook example	
FACAData1997	Real-world	US Federal Advisory Committee Act database	
Factory2000	Synthetic	Microsoft Access example	
Flav_R03-1	Real-world	US Department of Agriculture flavanoids database	
Flights	Synthetic	Textbook example	
FrenchTowns	Synthetic	PostgreSQL example	
GeoMetadb	Real-world	Gene expression database	
H1EFileFY2007	Real-world	US Visa applications	
Hydat	Real-world	Canadian water monitoring database	
Inventory	Synthetic	Textbook example	
Iso3166	Synthetic	PostgreSQL example	
IsoFlav_R2	Real-world	US Department of Agriculture isofalvone database	
iTrust	Synthetic	Medical application for teaching software testing methods	[78]
JWhoisServer	Real-world	Java-based Internet WHOIS server implementation	[26]
MozillaExtensions	Real-world	Extract from Firefox internet browser	
MozillaPermissions	Real-world	Extract from Firefox internet browser	
MozillaPlaces	Real-world	Extract from Firefox internet browser	
Mxm	Synthetic	MillionSong research data set of music meta-data	[12]
NistDML181	Real-world	NIST SQL conformance suite	[107]
NistDML182	Real-world	NIST SQL conformance suite	[107]
NistDML183	Real-world	NIST SQL conformance suite	[107]
NistWeather	Real-world	NIST SQL conformance suite	[107]
NistXTS748	Real-world	NIST SQL conformance suite	[107]
NistXTS749	Real-world	NIST SQL conformance suite	[107]
Northwind	Synthetic	Microsoft Access example	
Person	Synthetic	Textbook example	
ProductSales	Synthetic	Microsoft Access example	
Products	Synthetic	Textbook example	
RiskIt	Real-world	Insurance risk assessment application	[91]
Skype	Real-world	Voice-over-IP and messaging application	
SRAMetadb	Real-world	Gene sequencing database	
SongTrackMetadata	Synthetic	MillionSong research data set of music meta-data	[12]
StackOverflow	Real-world	Database previously used by a popular programming question and answer website	
StudentResidence	Synthetic	Textbook example	
TweetComplete	Synthetic	Teaching material	
UnixUsage	Real-world	Application to monitor and record use of UNIX commands	[91]
Usda	Synthetic	PostgreSQL example	
WordNet	Real-world	Graph visualiser for Wordnet	
World	Synthetic	MySQL sample	

Table 3.2: Attributes of parsed schemas in SchemaAnalyst

Schema	Tables	Columns	Checks	Foreign Keys	Not Nulls	Primary Keys	Uniques	$\sum$Constraints
AdmissionsPatient	3	17	0	0	0	0	3	3
ArtistSimilarity	2	3	0	2	0	1	0	3
ArtistTerm	5	7	0	4	0	3	0	7
BankAccount	2	9	0	1	5	2	0	8
BioSQL	28	129	0	52	86	24	24	186
BookTown	22	67	2	0	15	11	0	28
BrowserCookies	2	13	2	1	4	2	1	10
ChromeDB	2	7	0	0	5	2	2	9
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
Crafts2002	15	140	0	0	0	0	9	9
CustomerOrder	7	32	1	7	27	7	0	42
DavilaDjango	32	248	0	41	94	32	0	167
DHDBookstore	7	39	0	0	0	0	7	7
DellStore	8	52	0	0	39	0	0	39
Employee	1	7	3	0	0	1	0	4
Examination	2	21	6	1	0	2	0	9
FACAData1997	12	173	0	0	0	0	18	18
Factory2000	4	17	0	0	0	0	4	4
Flav_R03_1	7	50	0	0	0	0	6	6
Flights	2	13	1	1	6	2	0	10
FrenchTowns	3	14	0	2	13	0	9	24
GeoMetaDB	11	112	0	0	0	0	0	0
H1EFileFY2007	1	39	0	0	0	0	0	0
Hydat	33	407	0	0	0	0	32	32
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
IsoFlav_R2	6	40	0	0	0	0	5	5
iTrust	42	309	8	1	88	37	0	134
JWhoisServer	6	49	0	0	44	6	0	50
MozillaExtensions	6	51	0	0	0	2	5	7
MozillaPermissions	1	8	0	0	0	1	0	1
MozillaPlaces	11	68	0	0	10	10	5	25
Mxm	2	6	0	1	0	1	0	2
NistDML181	2	7	0	1	0	1	0	2
NistDML182	2	32	0	1	0	1	0	2
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	1	5	2	0	13
NistXTS748	1	3	1	0	1	0	1	3
NistXTS749	2	7	1	1	3	2	0	7
Northwind	15	95	0	0	33	14	0	47
Person	1	5	1	0	5	1	0	7
ProductSales	6	25	0	0	0	0	5	5
Products	3	9	4	2	5	3	0	14
RiskIt	13	57	0	10	15	11	0	36
Skype	6	31	0	0	0	0	0	0
SRAMetadb	11	288	0	0	0	3	0	3
SongTrackMetadata	1	14	0	0	0	1	0	1
StackOverflow	4	43	0	0	5	0	0	5
StudentResidence	2	6	3	1	2	2	0	8
TweetComplete	2	10	0	1	0	2	0	3
UnixUsage	8	32	0	7	10	7	0	24
Usda	10	67	0	0	31	0	0	31
WordNet	8	29	0	0	22	8	1	31
World	3	24	1	2	18	3	0	24
Total	398	2983	39	146	603	214	139	1141

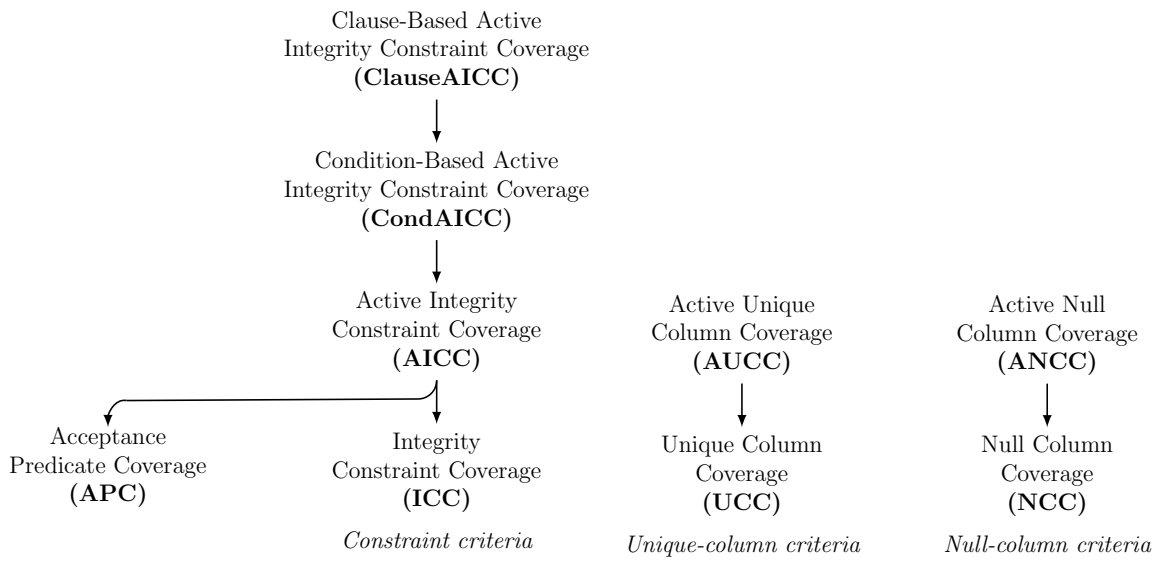


Figure 3.5: The subsumption hierarchies of coverage criteria for the testing of relational database schemas [76].

within a schema. These can be transformed into SQL `INSERT` statements using the *Input/Output* component in SchemaAnalyst. Data generation is implemented using a search-based approach and consisting of two main parts – a collection of coverage criteria and a set of search algorithms [76]. A coverage criterion specifies logically how the produced data should test the schema, expressed as a series of predicates. The search algorithms attempt to generate data to satisfy those predicates, which can then be output as `INSERT` statements and be executed with a DBMS. The following Sections provide a brief description of the main parts of the data generation component, however as these represent work not undertaken by myself as part of this thesis, the reader is referred elsewhere for further detail [76].

3.5.1 Coverage criteria

Given a schema containing some set of constraints expressed in the SchemaAnalyst intermediate representation of SQL, a coverage criterion formulates a series of predicates that describe a number of test cases, which should either be accepted or rejected by the database. The choice of criterion therefore determines which constraints in the schema are tested and how they are exercised. In SchemaAnalyst, there are three types of coverage

criteria available — constraint criteria, unique column criteria and null column criteria — which each produce predicates based on testing different element of the schema. These form three different subsumption hierarchies, as shown in Figure 3.5, where the more complex criteria towards the top of each hierarchy subsume those below them.

Constraint criteria

The five constraint criteria in SchemaAnalyst generate test predicates according to the constraints present in a given schema. The simplest, *Acceptance Predicate Coverage* (APC) criteria produces two predicates per table — specifying that the two resulting test cases should be accepted and rejected by the database, respectively. In the latter case this means that at least one constraint must be violated, although the predicate does not specify which. The *Integrity Constraint Coverage* (ICC) criteria improves upon APC by requiring two test cases be created for each integrity constraint in the schema, with the condition that one satisfies the constraint and the other violates it, ensuring each constraint is exercised. However, it does not guarantee that any data will be produced that will be accepted into the database as the predicates do not specify that the data produced to satisfy one constraint must also satisfy any other constraints. It is therefore possible that data produced to satisfy one constraint is in violation of another, and thus is rejected by the database. To alleviate this problem, *Active Integrity Constraint Coverage* (AICC) creates predicates that specify the given constraint under test must determine the overall acceptance or rejection of the data. The predicate therefore requires any other constraints in the same table are satisfied by the data generated, such that only the constraint under test may cause rejection. When generating data to test some types of constraints, it is possible that satisfying data can easily be generated by using the `NULL` value. For example, `NULL` is considered trivially unique by a `UNIQUE` constraint. However, such data may not be as useful as non-`NULL` unique values. The *Condition-Based Active Integrity Constraint Coverage* (CondAICC) criteria uses the same approach as AICC, but in addition specifies that tests should be produced to satisfy each constraint with and without `NULL` values, where applicable. Finally, the *Clause-Based Active Integrity Constraint Coverage* (ClauseAICC) criteria enhances CondAICC by additionally specifying that each individual clause in the constraint under test must be evaluated to true and false separately, whilst maintaining all other clauses. This ensures the selected clause determines whether the generated statement will be accepted or rejected. In effect,

ClauseAICC therefore leads to the production of tests that ensure each component part (e.g., each column) in a constraint is correct, and not the result of an error of commission, which overly constrains the data accepted into the database.

Unique column criteria

While the constraint criteria produce test data to ensure the correctness of existing constraints, the two unique column criteria test whether a `UNIQUE` constraint has been omitted. For the simpler *Unique Column Coverage* (UCC) criteria, this requires two test predicates per column — one containing unique column values, and one containing non-unique column values. As the value `NULL` is considered trivially unique, the predicates also specify the values used must be non-`NULL`. However, as with ICC, when a value generated by UCC is rejected by a database it is unclear whether this is due to an existing `UNIQUE` constraint, or some other constraint in the schema. Therefore, the *Active Unique Column Coverage* (AUCC) criteria extends the definition of UCC to stipulate that all constraints other than those already requiring uniqueness (i.e., `PRIMARY KEY` and `UNIQUE` constraints) must be satisfied.

Null column criteria

The two null column criteria produce tests containing `NULL` and non-`NULL` values, which help to identify the omission of `NOT NULL` constraints. The *Null Column Coverage* (NCC) criteria produces two predicates per column — one requiring a `NULL` value, the other a non-`NULL` value. As with UCC and ICC, when a generated statement is rejected it may either be due to a `NOT NULL` constraint, or some other existing constraint in the schema. The *Active Null Column Coverage* (ANCC) criteria resolves this by specifying that all other constraints that do not require a non-`NULL` value must be satisfied, such that only a `NOT NULL` can determine the acceptance or rejection.

3.5.2 Search algorithms

Once a series of predicates have been produced by some criterion, a search algorithm is applied to attempt to generate test data that satisfies the set of predicates for each test

case. There are currently two main search algorithms implemented in SchemaAnalyst — Random$^+$ and Alternating Variable Method (AVM).

Random$^+$

The Random$^+$ algorithm produces test data by randomly generating values of the correct data type for each of the columns needed in the test case. Once values for all required columns have been produced these are evaluated against the test predicate, to determine whether the data will be accepted or rejected by the database, as required in the predicate, once converted into an SQL `INSERT` statement. If the predicate is not met new values are generated, until either the predicate is met or some specified maximum number of values have been generated without success.

As some predicates may require the absence of data for one or more column (e.g., producing data that causes a `NOT NULL` constraint to reject the input), the search technique uses the value `NULL` instead of generating data at a frequency specified by a given probability. In addition, satisfying some constraints may require specific values that may be very unlikely to be generated at random (e.g., a `CHECK` constraint specifying a column of a string-like data type must be in a given list of values). Therefore, the Random$^+$ algorithm is augmented with a 'library' of values that have been extracted from the schema, for example constants specified in `CHECK` constraints. As with `NULL` values, these are used according to configured probability.

Alternating Variable Method (AVM)

Because the Random$^+$ algorithm uses randomly generated values it may require many repeated attempts to produce data that satisfies the required predicate, thus increasing the execution cost of data generation. The AVM algorithm attempts to reduce this by providing an alternative that uses a heuristic to guide the search to produce data that satisfies the test predicates. The search process begins by initialising each required column value to a default. Then, a change is made to one value and the data is evaluated using a fitness function, to determine whether that change has resulted in the data being "closer" to satisfying the predicate. If it has, the search continues with that change to

the data, otherwise a different change is applied. The search algorithm continues this process iteratively, until either data has been produced that satisfies the predicate or no change to the current values produces an improved fitness value, causing the values to be replaced with randomly generated values and the search restarted. To prevent this search process from continuing indefinitely (i.e., there may exist no possible values that satisfy all clauses of the predicate), the search is terminated after either a specified number of fitness evaluations or search restarts.

3.5.3 Fitness functions

Metaheuristic search algorithms, such as AVM, produce candidate solutions to a problem – in this case, generating data that satisfies a set of predicates produced by a coverage criterion – that must then be evaluated, to determine how successful those solutions are. In the case of the Random$^+$ algorithm, success is considered in a binary fashion, with a new solution being generated if the previous one is unsuccessful at satisfying the predicates. In contrast, the AVM algorithm must differentiate between different degrees of success, as it produces the next candidate solution by modifying whichever previous solution is nearest to being successful. This is calculated according to a fitness function that provides a fitness score between 0 and 1 – where 0 is a successful solution to the current problem – that can be used as a measurement of the "distance" between the proposed solution and a successful one. The AVM algorithm searches the domain of inputs to a problem by minimising this distance, until a suitable value is found or a termination criteria is met.

In the SchemaAnalyst tool, fitness functions are automatically generated based upon the predicates produced by the coverage criterion. These are then used to evaluate each possible set of data produced by the AVM algorithm. The fitness functions are formed from the distance functions in Figure 3.6, using `value_dist` to compare two values. This applies either an *atomic* or *compound* distance measurement, depending on the type of data being tested. Atomic data types are those that can be represented by a single value, such as numeric (e.g., `INTEGER`, `DOUBLE`, `NUMERIC`) and boolean values. Compound data types are represented as multiple atomic data values. For example date (e.g., `DATE`, `DATETIME`) values are stored as a series of integers for each component, while character values (e.g., `CHAR`, `VARCHAR`) consist of a variable-length list of integers, to allow for

value_dist(a, op, b)

a	b	
`NULL`	`NULL`	return 0
`NULL`	*any*	return 1
any	`NULL`	return 1
Atomic	Atomic	return $norm(atomic\_dist(a, op, b))$
Compound	Compound	return $norm(compound\_dist(a, op, b))$

atomic_dist(a, op, b)

op	
$=$	if $(\|a-b\| = 0)$ then return 0 else return $\|a-b\| + 1$
$\neq$	if $(\|a-b\| \neq 0)$ then return 0 else return 1
$<$	if $(a - b < 0)$ then return 0 else return $(a - b) + 1$
$\leq$	if $(a - b \leq 0)$ then return 0 else return $(a - b) + 1$
$>$	if $(b - a < 0)$ then return 0 else return $(b - a) + 1$
$\geq$	if $(b - a \leq 0)$ then return 0 else return $(b - a) + 1$

compound_dist$((a_1, \ldots a_p), op, (b_1, \ldots b_q))$

op	
$=$	return $\|p - q\| + \sum_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, =, b_i))$
$\neq$	if $(p \neq q)$ then return 0 else return $min_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, \neq, b_i))$

other	$d \leftarrow 0$
	while $(i \leq min(p, q) \wedge d = 0)$
	  if $(a_i \neq b_i)$ then $d \leftarrow norm(atomic\_dist(a_i, op, b_i))$
	  $i \leftarrow i + 1$
	end while
	return $d + atomic\_dist(p, op, q)$

and_dist$(d_1, \ldots, d_n)$
return $norm(\sum_{i=0}^{i=n} d_i)$

or_dist$(d_1, \ldots, d_n)$
return $min_{i=0}^{i=n} d_i$

Figure 3.6: The distance functions used in the SchemaAnalyst tool to produce a fitness function for search-based data generation [76].

their possibly unspecified size. The atomic and compound distance functions for a given predicate are combined using the `and_dist` and `or_dist` functions, as applicable, and are each normalised to an equal weighting of the overall fitness function. The generated function can then be used by the AVM algorithm to guide the search algorithm to locate data values that satisfy the predicates produced by the coverage criterion for the current test case.

3.6 Mutation Framework

In order to simplify the definition of mutation operators for SQL artefacts (e.g., schemas) within SchemaAnalyst, such as the 14 integrity constraint mutation operators formally described in Chapter 4, I have created a *mutation framework* within the tool. This encapsulates much of the behaviour shared between different operators to reduce the duplication of code, therefore reducing the likelihood of faults in the mutation analysis process itself. The process of mutation requires the following steps to be taken:

1. Traverse the object graph of the abstract representation produced from parsing the schema, to find a point where a mutation can be applied.
2. Make a duplicated copy of the schema prior to making a modification, to ensure both that the original schema is left unmodified and that each mutant only contains a single injected fault.
3. Apply the change to the duplicated copy of the component that is being mutated, according to the implementation of the mutation operator.

The mutation framework, which forms the *Mutation* component of SchemaAnalyst, implements this functionality within the `mutation` package and is divided into four main sub-packages, as shown in Figure 3.7 – `mutator`, `supplier`, `pipeline` and `analysis`. Together these classes are used to create a `Mutant` copy of some artefact, which has been modified to model a programmer fault, by executing a `MutantProducer`, that defines how to inject theses fault into the artefact.

3.6.1 The `mutator` package

Although there are many different types of constraint that may be mutated, the actual type of change being made can be categorised as one of a limited list of modification types. Each of these is implemented as a `Mutator` class that can manipulate any type of component, such that they can be reused regardless of the type constraint being mutated. For example, the list of columns specified for a `PRIMARY KEY` constraint or a `UNIQUE`

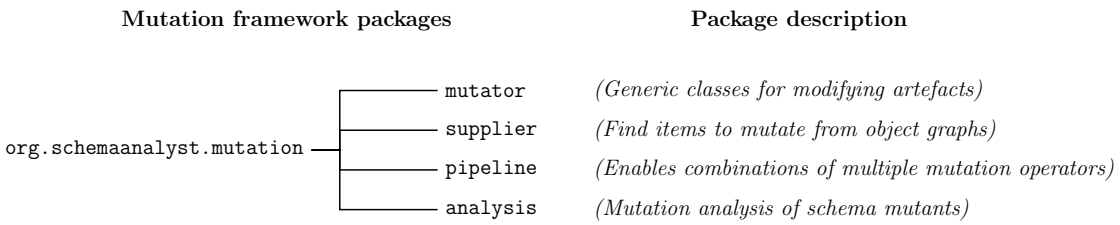


Figure 3.7: The packages of the SchemaAnalyst mutation framework with descriptions of their purpose.

constraint can be modified using the same classes, as both are defined by the lists of columns they are defined for.

Initially, four `Mutator` classes were implemented with the mutation framework. Firstly, `ListElementAdder` modifies a list of components by adding each value from a list of valid additional values in turn – so, given the existing list (a, b) and valid additional values (c, d), this class will produce two mutants with the lists (a, b, c) and (a, b, d). Secondly, the `ListElementRemover` removes each component from a list in turn, for example modifying the list (a, b, c) to produce the mutated lists (b, c), (a, c) and (b, c). Next, the `ListElementExchanger` mutator alters a list by replacing each component in turn with an component from another list of alternatives. For example, applying this mutator to the list (a, b) with the list of alternative (c, d) will produce four mutated lists – (c, b), (d, b), (a, c) and (a, d). Finally, `ElementNullifier` modifies an component by replacing it with `null`, which when used with the SchemaAnalyst representation of SQL as removing it entirely. In addition to these mutators, during the development of the operators discussed in Chapter 4 an additional `RelationalOperatorExchanger` mutator was added that exchanges a relational operator (as used in a `CHECK` constraint) with each other operator ($=$,$\neq$,$<$,$\leq$,$\geq$). This made it simpler to define the `CRelOpE` mutation operator, formally described later in that Chapter, therefore reducing the likelihood of a mistake in the implementation.

3.6.2 The `supplier` package

As the `Mutator` classes are designed to be unaware of the type of artefact they are mutating, it is necessary to combine them with a set of classes that extract components

Table 3.3: Description of the `Supplier` classes implemented in the mutation framework.

Supplier	Description
`PrimaryKeyConstraintSupplier`	Supplies `PRIMARY KEY` constraints from each table of a schema
`PrimaryKeyColumnSupplier`	Supplies a list of columns from a `PRIMARY KEY` constraint
`PrimaryKeyColumnsWithAlternativesSupplier`	Supplies a list of columns from a `PRIMARY KEY` constraint and a list of columns not in the constraint from the same table
`UniqueConstraintSupplier`	Supplies `UNIQUE` constraints from each table of a schema
`UniqueColumnSupplier`	Supplies a list of columns from a `UNIQUE` constraint
`UniqueColumnsWithAlternativesSupplier`	Supplies a list of columns from a `UNIQUE` constraint and a list of columns not in the constraint from the same table
`ForeignKeyConstraintSupplier`	Supplies `FOREIGN KEY` constraints from each table of a schema
`ForeignKeyColumnSupplier`	Supplies a list of (local column, reference column) pairs from a `FOREIGN KEY` constraint
`ForeignKeyColumnPairWithAlternativesSupplier`	Supplies a list of (local column, reference column) pairs from a `FOREIGN KEY` constraint and a list of (local column, reference column) pairs not in the constraint from the same tables
`CheckConstraintSupplier`	Supplies `CHECK` constraints from each table of a schema
`CheckExpressionSupplier`	Supplies expressions from `CHECK` constraints
`ExpressionSupplier`	Supplies each of a given type of descendant expression from a root expression
`RelationalExpressionSupplier`	Supplies each relational expressions from a root expression
`RelationalExpressionOperatorSupplier`	Supplies the relational operator from a relational expression
`InExpressionRHSListSupplier`	Supplies each `IN` expression where the right hand-side is a list (e.g., '`a IN (x,y,z)`') from a root expression
`InExpressionRHSListExpressionSupplier`	Supplies the list from the right hand-side of an `IN` expression

from a schema that they can be applied to, in order to produce mutants. In the mutation framework this functionality is implemented as a series of `Supplier` classes. These also include the logic required to produce duplicated versions of the schema being mutated, such that a new duplicate of the schema can be created prior to a modification being made by a mutator. This ensures that each mutant only contains a single mutation and the original schema is left unchanged.

Each supplier class accepts some artefact, `A`, as an input and extracts one or more component, `C`, from it, producing multiple mutants by iterating through each component in turn and passing it to the `Mutator` class specified in the operator. A number of the `Supplier` classes implemented are listed and described in Table 3.3. To avoid repetition in various supplier the mutation framework supports the chaining of multiple suppliers together using a `LinkedSupplier` if the first accepts $\texttt{A}_1$ and produces $\texttt{C}_1$, the second accepts $\texttt{A}_2$ and produces $\texttt{C}_2$, and $\texttt{C}_1 = \texttt{A}_2$. For example, the `PrimaryKeyConstraintSupplier` class can be used to extract `PRIMARY KEY` constraints from each table of any schema (`A` = `Schema`, `C` = `PrimaryKeyConstraint`). This can then either be chained with the `PrimaryKeyColumnSupplier` to provide a list of columns that can be removed by the `ListElementRemover` mutator, or the `PrimaryKeyColumnsWithAlternativesSupplier` to provide lists of both the columns already in the constraint and columns not in the con-

straint for use with the `ListElementAdder` or `ListElementExchanger` mutators. When suppliers are chained in this way, mutants are produced by extracting all components in the bottom level supplier before iterating to the next value from the top level supplier. The process is then repeated until both suppliers are exhausted, in a fashion similar to a depth-first search process. In addition, as `LinkedSupplier` is itself an instance of `Supplier` it is possible to chain together any number of these together, provided each supplier in the chain yields an output type that matches the input type of the next, to allow components to be extracted from arbitrarily complex artefacts.

3.6.3 The `pipeline` package

By combining the mutators and suppliers described above various mutation operators can be defined, which can each produce a series of mutant schemas when executed with a schema. For mutation analysis it is necessary to execute multiple operators to produce a set of mutants that model a wide range of faults. This is implemented in the mutation framework as a `Pipeline`, which applies a series of operators in sequence to generate the full set of mutants. As discussed later in Chapter 6, a pipeline may also include classes that are used to remove certain types of undesirable mutants after they have been generated but prior to mutation analysis.

3.6.4 The `analysis` package

Once a mutation pipeline has been specified this can be used to generate a set of mutants that can be used for mutation analysis, the code for which is implemented in the `analysis` package of the mutation framework. The classes in this package coordinate the generation of test data (by invoking the data generation component of SchemaAnalyst), production of mutants, execution of the data against both the original schema and each mutant schema, and the comparison of the results for the mutants to determine whether they have been killed. The exact technique used to analyse the mutants can be varied prior to execution, using various optimised implementations discussed later in Chapter 7.

3.7 Summary

This Chapter described a number of different parts of the SchemaAnalyst tool, including the existing SQL representation and data generation components, as well as the mutation framework that forms a contribution of this thesis. All of the remaining Chapters of this thesis now make use of these components to explore various facets of mutation analysis for relational database schemas. Firstly, Chapter 4 makes use of the mutator, supplier and pipeline classes of the mutation framework to define a series of mutation operators for mutating the integrity constraints of schemas. Then, Chapter 5 investigates the various configurations of the data generation component to determine which can kill the greatest proportion of those mutants, to identify the data generator and coverage criterion that produces test data with the highest fault-finding capability. Following this, Chapter 6 describes three types of mutant that reduce either the effectiveness or efficiency of mutation analysis, or both. By describing patterns that can be detected in the SchemaAnalyst intermediate representation of SQL, these are automatically removed using a series of classes that can be added to a mutation pipeline. Finally, Chapter 6 explores how mutation analysis optimisations for programs from the literature can inspire a collection of techniques that attempt to reduce the computational cost of mutation analysis for schemas. Each of these are then implemented as part of the `analysis` package of the mutation framework and evaluated through an empirical experiment.

Chapter 4

Mutation Operators for Relational Database Schemas

4.1 Introduction

Utilising the mutation framework developed as part of the SchemaAnalyst tool, discussed in Chapter 3, this Chapter now describes a set of 14 mutation operators for generating mutants of relational database schemas. These modify a range of different integrity constraints expressed within schemas with each modelling a different possible mistake made by the author when designing the schema, such that a wide range of potential faults can be identified. In some cases, static analysis must be applied to detect type-compatibility of columns used in mutated constraints, to ensure syntactic validity. These mutants can be used as part of a mutation analysis experiment to estimate the fault-finding capability of a given test suite – in this case, a series of SQL `INSERT` statements. The operators have been previously discussed as part of published works produced during my PhD [62, 118, 76], with the formal definitions and descriptions of the productivity of each (i.e., how many mutants they produce) given in this Chapter providing a greater level of detail, to form a contribution of this thesis.

The Chapter begins by explaining the approach used to classify each mutation operator and the operator naming scheme based upon this, followed by the description of a

number of utility functions that are used for the remainder of the Chapter in the algorithmic definitions of the mutation operators. The 14 operators are then detailed, including original definitions of the algorithms I have used to implement them in the SchemaAnalyst tool, as well as numerous examples. Finally, the productivity of each operator is specified formulaically in terms of the various columns and constraints contained within the schema under mutation.

The contributions this Chapter makes are:

1. Textual and algorithmic descriptions of all 14 mutation operators for relational database schemas, specifying the programmatic mechanism by which they produce mutants, including worked examples of how these apply to simple schemas; and

2. Novel definitions of the productivity of each of the 14 operators, used to calculate how many mutants will be produced by a given schema, in terms of the attributes of a schema under test.

4.2 Preliminaries

This Section provides some background information that is useful for understanding the rest of this Chapter. Firstly, a scheme for classifying and naming relational database schema mutation operators is described. This can be used to derive a simple descriptive name for each operator that include the constraint affected and the type of modification made. Secondly, a collection of utility functions are described, which are used in the algorithm definitions of the mutation operators. These are approximately equivalent to methods implemented in SchemaAnalyst either as part of the intermediate representation of SQL (Section 3.3) or the mutation framework (Section 3.6).

4.2.1 Operator classification and naming schema

To enable easy identification of the mutation operators, each is assigned a name comprising of two parts – the kind of constraint that it can mutate, and the type of modification it makes to those constraints. These two categories can take the following possible values:

Constraints:

1. `PRIMARY KEY`
2. `FOREIGN KEY`
3. `UNIQUE`
4. `NOT NULL`
5. `CHECK`

Modifications:

1. Addition
2. Removal
3. Exchange

An operator with an *addition* modification adds an element (e.g., a column) to an existing constraint, while the *removal* modification removes an element from a constraint – or a constraint in its entirety. Finally, an *exchange* modification replaces part of an existing constraint (e.g., replacing a reference to a column in a constraint with the reference to another).

By combining the kind of constraint mutated and the type of modification made, a unique descriptive name can be produced for an operator. For example, using his scheme an operator mutating a `PRIMARY KEY` constraint by adding columns is given the name `PrimaryKeyColumnAddition`, which can be abbreviated to `PKColumnA`. Similarly, an operator mutating a `UNIQUE` constraint by removing columns is referred to as the `UniqueConstraintColumnRemoval` operator, or `UColumnR` for short. This naming scheme is used for the rest of this Chapter for referring to each mutation operator.

4.2.2 Utility functions for operator algorithms

The functions listed in Figure 4.1 are used to simplify the definition of the mutation operators later in Section 4.3. These are divided into decomposition functions, which extract information from a schema; composition functions, which produce components of a schema; and predicate functions, which are boolean tests executed against a schema. These correspond to functionality implemented in either the intermediate representation of SQL or in the mutation framework, both parts of the SchemaAnalyst tool. The exact details of these functions are omitted as an implementation detail.

▷ Decomposition functions

function getTables(*schema*) ▷ Set of tables in *schema*
function getTable(*c*) ▷ The table some constraint *c* relates to
function getColumns(*table*) ▷ Set of columns in *table*
function getColumns(*constraint*) ▷ Set of columns in *constraint*
function getPrimaryKey(*table*) ▷ The PRIMARY KEY of *table*
function getForeignKeys(*table*) ▷ Set of FOREIGN KEYs in *table*
function getForeignKey(*table*,*i*) ▷ The *i*th FOREIGN KEY in *table*
function getRefColumns(*fk*) ▷ Set of referenced columns in *fk*
function getRefTable(*fk*) ▷ The referenced table in *fk*
function getForeignKeyPairs(*fk*) ▷ The set of local-reference pairs in *fk*
function getUniques(*table*) ▷ Set of UNIQUEs in *table*
function getUnique(*table*,*i*) ▷ The *i*th UNIQUE in *table*
function getNotNulls(*table*) ▷ Set of NOT NULL in *table*
function getChecks(*table*) ▷ Set of CHECKS in *table*
function getCheck(*table*,*i*) ▷ The *i*th CHECKS in *table*
function getExpressions(*check*) ▷ Extracts expressions from *check*
function getExpressionList(*check*) ▷ Extracts list from IN LIST type *check*
function getExpressionRelOp(*check*) ▷ Extracts relational operator from REL OP type *check*
function getType(*column*) ▷ Data type of *column*

▷ Composition functions

function createMutant(*a*,*b*) ▷ Mutant of schema with *a* replaced by *b*
function createPrimaryKey(*cols*) ▷ Creates PRIMARY KEY over *cols*
function createForeignKey(*cols*,*refs*) ▷ Creates FOREIGN KEY over *cols* and *refs*
function createUnique(*cols*) ▷ Creates UNIQUE over *cols*
function createNotNull(*cols*) ▷ Creates NOT NULL for *cols*
function createCheck(*expressions*) ▷ Creates CHECK with *expressions*

▷ Predicate functions

function containsInListExpr(*expression*) ▷ Whether *expression* is IN LIST type
function containsRelOpExpr(*expression*) ▷ Whether *expression* is relation operation type

Figure 4.1: The utility functions, as implemented in the SchemaAnalyst tool, used to analyse and produce mutants of relational database schemas.

4.3 Operator Definitions

In this Section each operator is described in turn, grouped according to the kind of constraint it mutates. In each case, an algorithm listing is provided to define the exact behaviour, according to the functions of Figure 4.1.

4.3.1 `PRIMARY KEY` operators

Algorithm 1 Primary key column addition operator

```
function PKCColumnA(schema)
    mutants ← ∅
    for t in getTables(schema) do
        pk ← getPrimaryKey(t)
        for c in getColumns(t) do
            if c ∉ getColumns(pk) then
                mutant ← createPrimaryKey(getColumns(pk) ∪ {c})
                mutants ← mutants ∪ {createMutant(pk,mutant)}
    return mutants
```

PKColumnA: The function in Algorithm 1 describes the approach of the `PRIMARY KEY` column addition (PKColumnA) operator, which models the faults of either an omitted single-column `PRIMARY KEY` or omission of a column from an existing `PRIMARY KEY`. Mutants are produced by adding each column of a table to a `PRIMARY KEY`, if it is not already part of one. For example, with a table `x`:

```
x (a INT, b INT, PRIMARY KEY(a))
```

...which contains a single column `PRIMARY KEY`, one mutant will be produced, m_1, by adding the column `b`:

m_1: `x (a INT, b INT, PRIMARY KEY(a,b))`

If no `PRIMARY KEY` is defined for the table this operator will produce mutants with one added, containing each column in turn. So, for a table `y`:

```
y (a INT, b INT)
```

...which contains no `PRIMARY KEY`, the mutants m_2 and m_3 would be generated:

m_2: `y (a INT, b INT, PRIMARY KEY(a))`

m_3: `y (a INT, b INT, PRIMARY KEY(b))`

Algorithm 2 Primary key column removal operator

```
function PKCColumnR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        pk ← getPrimaryKey(t)
        for c in getColumns(pk) do
            mutant ← createPrimaryKey(getColumns(pk) \ {c})
            mutants ← mutants ∪ {createMutant(pk,mutant)}
    return mutants
```

PKColumnR: The function in Algorithm 2 describes the approach of the `PRIMARY KEY` column removal (PKColumnR) operator. This operator creates mutants by removing each column of existing `PRIMARY KEY` constraints in turn. For example, for the table `x`:

```
x (a INT, b INT, PRIMARY KEY(a,b))
```

...which contains a multi-column `PRIMARY KEY`, mutants $\texttt{m}_1$ and $\texttt{m}_2$ would be produced by removing `a` and `b` from the constraint, respectively:

$\texttt{m}_1$: `x (a INT, b INT, PRIMARY KEY(b))`

$\texttt{m}_2$: `x (a INT, b INT, PRIMARY KEY(a))`

If no columns remain in the constraint after removal (i.e., in the original schema it contained only a single column, which was removed) then the entire constraint is removed, as a `PRIMARY KEY` must be defined over at least one column.

Algorithm 3 Primary key column exchange operator

```
function PKCColumnE(schema)
    mutants ← ∅
    for t in getTables(schema) do
        pk ← getPrimaryKey(t)
        for c in getColumns(t) do
            if c ∉ getColumns(pk) then
                for pkc in getColumns(pk) do
                    mutant ← createPrimaryKey(c ∪ ( getColumns(pk) \ {pkc})
                    mutants ← mutants ∪ {createMutant(pk,mutant)}
    return mutants
```

PKColumnE: The function in Algorithm 3 describes the approach of the `PRIMARY KEY` constraint column exchange (PKColumnE) operator. Mutants are produced with this operator by exchanging each column in a `PRIMARY KEY` with each column in the same table that isn't already in the constraint. For example, given the table `x`:

```
x (a INT, b INT, c INT, PRIMARY KEY(a,b))
```

...which contains a multi-column `PRIMARY KEY`, mutants would be produced by exchanging the column `c`, the only column not already in the constraint, for the columns in the constraint, `a` and `b`, in turn. This yields two mutants, $\texttt{m}_1$ and $\texttt{m}_2$:

$\texttt{m}_1$: `x (a INT, b INT, c INT, PRIMARY KEY(c,b))`

$\texttt{m}_2$: `x (a INT, b INT, c INT, PRIMARY KEY(a,c))`

These mutants model a programmer mistakenly specifying the wrong column in a `PRIMARY KEY` constraint, but the right number of columns in total.

4.3.2 `FOREIGN KEY` operators

Algorithm 4 Foreign key column pair addition operator

```
function FKCColumnPairA(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for fk in getForeignKeys(t) do
            for x, y in getColumns(t), getColumns(getRefTable(fk)) do
                if getType(x) = getType(y) ∧ x ∉ getColumns(fk) ∧ y ∉ getRefColumns(fk) then
                    mutant ← createForeignKey(getColumns(fk) ∪ {x}, getRefColumns(fk) ∪ {y})
                    mutants ← mutants ∪ {createMutant(fk,mutant)}
    return mutants
```

FKColumnPairA: The function in Algorithm 4 describes the approach of the `FOREIGN KEY` column pair addition (FKColumnPairA) operator. Unlike other addition operators that select a single column to add, this operator must add a column from both the "local" table (where the constraint is defined) and the "reference" table (which the constraint refers to). This is because a `FOREIGN KEY` is expressed as a mapping between pairs of columns from two tables. It is also necessary to ensure that the two columns selected are of the same data type — otherwise the mutant created may not be syntactically valid. This requires static analysis of the schema to determine which columns have matching data types, and therefore make a valid addition to a `FOREIGN KEY` constraint. For every pair of columns from the local and reference tables that do match data type, and are not already included in the constraint, this operator will create a mutant with these added to an existing `FOREIGN KEY` constraint. For example, given the tables `x` and `y`:

```
x (a INT, b INT, c VARCHAR)
y (d INT, e INT, f VARCHAR, FOREIGN KEY (d) REFERENCES x(a))
```

...there are the following matching column pairs, according to their data type:

(1) `(d,a)` (2) `(e,a)` (3) `(d,b)` (4) `(e,b)` (5) `(f,c)`

Of these pairs (1), (2) and (3) use columns already in the `FOREIGN KEY` constraint, and are therefore excluded from the mutant generation. However, pairs (4) and (5) do not and are therefore used by the operator to produce the mutants $\texttt{m}_1$ and $\texttt{m}_2$:

$\texttt{m}_1$:
```
x (a INT, b INT, c VARCHAR)
y (d INT, e INT, f VARCHAR, FOREIGN KEY (d,e) REFERENCES x(a,b))
```

$\texttt{m}_2$:
```
x (a INT, b INT, c VARCHAR)
y (d INT, e INT, f VARCHAR, FOREIGN KEY (d,f) REFERENCES x(a,c))
```

These mutants model the programmer error of omitting a pair of columns from a `FOREIGN KEY` constraint, or omitting the constraint entirely.

Algorithm 5 Foreign key column pair removal operator

```
function FKCColumnPairR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for fk in getForeignKeys(t) do
            for a, b in cols, colsref do
                mutant ← createForeignKey(getColumns(fk) \ {a}, getRefColumns(fk) \ {b})
                mutants ← mutants ∪ {createMutant(fk,mutant)}
    return mutants
```

FKColumnPairR: The function in Algorithm 5 describes the approach of the `FOREIGN KEY` constraint column pair removal (FKColumnPairR) operator. This operator creates mutants by removing pairs of columns from each `FOREIGN KEY` constraint in the schema, if any exist, where a pair consists of a column in the "local" table and the corresponding column from the reference table. If the removal of a pair leaves the `FOREIGN KEY` with no remaining column pairs (i.e., if the original constraint only consisted of a single pair) then the constraint is removed entirely. For example, given the tables `x` and `y`:

```
x (a INT, b INT)
y (c INT, d INT, FOREIGN KEY (c,d) REFERENCES x(a,b))
```

...there are two local-reference column pairs, `(c,a)` and `(d,b)`. These are removed in turn to create two mutants, $\texttt{m}_1$ and $\texttt{m}_2$, which model the programmer mistake of erroneously including an extra pair of columns in a `FOREIGN KEY`:

```
m1:  x (a INT, b INT)
     y (c INT, d INT, FOREIGN KEY (d) REFERENCES x(b))

m2:  x (a INT, b INT)
     y (c INT, d INT, FOREIGN KEY (c) REFERENCES x(a))
```

Algorithm 6 Foreign key column pair exchange operator

```
function FKCColumnPairE(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for fk in getForeignKeys(t) do
            for a, b in getColumns(fk), getRefColumns(fk) do
                for x, y in getColumns(t), getColumns(getRefTable(fk)) do
                    if ((x = a ∧ y ≠ b) ∨ (x ≠ a ∧ y = b)) ∧ getType(x) = getType(y) then
                        cols ← ( getColumns(fk) \{a} ) ∪ {x}
                        refs ← ( getRefColumns(fk) \{b} ) ∪ {y}
                        mutant ← createForeignKey(cols, refs)
                        mutants ← mutants ∪ {createMutant(fk,mutant)}
    return mutants
```

FKColumnPairE: The function in Algorithm 6 describes the approach of the `FOREIGN KEY` constraint column pair exchange (FKColumnPairE) operator. This operator exchanges columns from the pairs specified in existing `FOREIGN KEY` constraint, for example replacing the constraint `FOREIGN KEY (a) REFERENCES t(x)` with `FOREIGN KEY (a) REFERENCES t(y)` or `FOREIGN KEY (b) REFERENCES t(x)`. Note that as mutation operators should only make a single change, only one column reference is changed. As with the FKColumnPairA operator, the columns added to the constraint must have matching types. Therefore, the schema is statically analysed to determine which two of columns — one from each the "local" and "reference" tables — form a valid pair to use in the constraint. Mutants are then produced by exchanging the existing columns in the `FOREIGN KEY` constraint with the new column pair. This operator therefore models the programmer error of mispecifying one of the columns in a `FOREIGN KEY` column pair.

As an example, given the tables `x` and `y` with a single column `FOREIGN KEY` constraint:

```
x (a INT, b INT, c VARCHAR, d VARCHAR)
y (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e) REFERENCES x(a))
```

...according to the constraints of matching data types and modifying only one column reference, 2 mutants can be produced — m_1 to m_2:

```
m1: x (a INT, b INT, c VARCHAR, d VARCHAR)
    y (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e) REFERENCES x(b))

m2: x (a INT, b INT, c VARCHAR, d VARCHAR)
    y (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (f) REFERENCES x(a))
```

Alternativly, where there are multiple column pairs in the `FOREIGN KEY` constraint, consider the example tables `m` and `n`:

```
m (a INT, b INT, c VARCHAR, d VARCHAR)
n (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e,g) REFERENCES m(a,c))
```

...then applying the mutation FKCColumnPairA operator would generate 4 mutants — m_1 to m_4:

```
m1: m (a INT, b INT, c VARCHAR, d VARCHAR)
    n (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e,g) REFERENCES m(b,c))

m2: m (a INT, b INT, c VARCHAR, d VARCHAR)
    n (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (f,g) REFERENCES m(a,c))

m3: m (a INT, b INT, c VARCHAR, d VARCHAR)
    n (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e,g) REFERENCES m(a,d))

m4: m (a INT, b INT, c VARCHAR, d VARCHAR)
    n (e INT, f INT, g VARCHAR, h VARCHAR, FOREIGN KEY (e,h) REFERENCES m(a,c))
```

4.3.3 `UNIQUE` constraint operators

UColumnA: The function in Algorithm 7 describes the approach of the `UNIQUE` constraint column addition (UColumnA) operator. This produces mutants in two phases. Firstly, for each column in a table, a mutant is created with a `UNIQUE` constraint added to that column, provided one doesn't already exist. Secondly, for each `UNIQUE` constraint already in the original schema, a mutant is created with each column in the table added to that constraint, provided that column is not already included. Applying this operator to the table `x`:

Algorithm 7 Unique constraint column addition operator

```
function UCColumnA(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for c in getColumns(t) do                          ▷ Produce new UNIQUE constraints
            if getUnique(c) ∉ getUniques(t) then
                mutant ← createUnique(c)
                mutants ← mutants ∪ {createMutant(∅,mutant)}
        for uc in getUniques(t) do                         ▷ Modify existing UNIQUE constraints
            for c in getColumns(t) do
                if c ∉ getColumns(uc) then
                    mutant ← createUnique(getColumns(uc) ∪ {c})
                    mutants ← mutants ∪ {createMutant(uc,mutant)}
    return mutants
```

```
x (a INT, b INT, c INT, UNIQUE(a))
```

...would produce two mutants in the first phase, $\texttt{m}_1$ and $\texttt{m}_2$, by adding `UNIQUE` constraints for the columns `b` and `c`. A constraint is not added for `a` as it is already subject to a `UNIQUE` constraint. These mutants model the programmer error of omitting a `UNIQUE` constraint from the schema. The produced mutants are as follows:

```
m1:  x (a INT, b INT, c INT, UNIQUE(a), UNIQUE(b))

m2:  x (a INT, b INT, c INT, UNIQUE(a), UNIQUE(c))
```

In the second phase, columns are added to any existing constraints, modelling the omission of a column from a `UNIQUE` that is already part of the schema. In this case, `b` and `c` are individually added to the `UNIQUE` constraint to produce two mutants, $\texttt{m}_3$ and $\texttt{m}_4$:

```
m3:  x (a INT, b INT, c INT, UNIQUE(a,b))

m4:  x (a INT, b INT, c INT, UNIQUE(a,c))
```

Algorithm 8 Unique constraint column removal operator

```
function UCColumnR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for uc in getUniques(t) do
            for c in getColumns(uc) do
                mutant ← createUnique(getColumns(uc) \ {c})
                mutants ← mutants ∪ {createMutant(uc,mutant)}
    return mutants
```

UColumnR: The function in Algorithm 8 describes the approach of the `UNIQUE` constraint column removal (UColumnR) operator. Applying this operator creates mutants by removing a single column from each `UNIQUE` constraint in turn – either producing a constraint with fewer columns if the original had multiple columns, or removing the constraint entirely if the original only had a single column. For example, with the table `x`:

```
x (a INT, b INT, c INT, UNIQUE(a,b), UNIQUE(c))
```

...this operator would be applied twice – once per `UNIQUE` constraint. Mutating the first constraint would create two mutant schemas, $\texttt{m}_1$ and $\texttt{m}_2$, by removing each column from the constraint in turn, whilst leaving the second constraint unaltered:

$\texttt{m}_1$: `x (a INT, b INT, c INT, UNIQUE(b), UNIQUE(c))`

$\texttt{m}_2$: `x (a INT, b INT, c INT, UNIQUE(a), UNIQUE(c))`

Next, the operator would change the second `UNIQUE` constraint. In this case, the constraint is removed entirely, as there is only one column to remove, to give $\texttt{m}_3$:

$\texttt{m}_3$: `x (a INT, b INT, c INT, UNIQUE(a,b))`

All three of these mutants model the mistake of the programmer including too many columns in a `UNIQUE` constraint, thus constraining the data accepted into the database too tightly.

Algorithm 9 Unique constraint column exchange operator

```
function UCColumnE(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for uc in getUniques(t) do
            for c in getColumns(t) do
                if c ∉ getColumns(uc) then
                    for ucc in getColumns(uc) do
                        mutant ← createUnique(c ∪ ( getColumns(uc) \ {ucc})
                        mutants ← mutants ∪ {createMutant(uc,mutant)}
    return mutants
```

UColumnE: The function in Algorithm 9 describes the approach of the `UNIQUE` constraint column exchange (UColumnE) operator. Each mutant generated by this operator is created by exchanging a single column in an existing `UNIQUE` constraint with another column from the same table, provided that column is not already part of the constraint. So, given the table `x`:

```
x (a INT, b INT, c INT, UNIQUE(a))
```

...the column `a` in the `UNIQUE` constraint would be exchanged with the two columns not in the constraint, `b` and `c`, to produce two mutants, $\texttt{m}_1$ and $\texttt{m}_2$:

$\texttt{m}_1$: `x (a INT, b INT, UNIQUE(b))`

$\texttt{m}_2$: `x (a INT, b INT, UNIQUE(c))`

In this respect, the UColumnE operator functions much like the PKColumnE operator. However, `UNIQUE` constraints differ from `PRIMARY KEY` constraints, in that more than one may be defined per table. In this instance, each `UNIQUE` constraint is treated separately, so any column not included in a constraint may be exchanged into the constraint, even if it is part of another `UNIQUE` constraint.

This leads to the possibility of two `UNIQUE` constraints over the same columns — suppose columns($\texttt{UC}_1$) = { `a` } and columns($\texttt{UC}_2$) = { `b` }, then exchanging `a` with `b` in $\texttt{UC}_1$ would mean columns($\texttt{UC}_1$) = columns($\texttt{UC}_2$). As two `UNIQUE` constraints with matching columns do not confer any semantic difference than only one `UNIQUE` constraint over the same columns only one is retained, effectively removing the mutated constraint. For example, given the table `y`:

```
y (a INT, b INT, UNIQUE(a), UNIQUE(b))
```

...there are two single-column `UNIQUE` constraints to mutate. As `b` is not already part of the first constraint, it is exchanged with `a` to give `UNIQUE(b)`, which is discarded to give the mutant $\texttt{m}_1$ below. Likewise, `a` can be exchanged with `b` in the second `UNIQUE` constraint, giving a duplicated constraint that is discarded to give the mutant $\texttt{m}_2$ below:

$\texttt{m}_1$: `y (a INT, b INT, UNIQUE(b))`

$\texttt{m}_2$: `y (a INT, b INT, UNIQUE(a))`

4.3.4 `NOT NULL` constraint operators

Algorithm 10 Not null addition operator

```
function NNCA(schema)
    mutants ← ∅
    for t in getTables(schema) do
        nns ← getNotNulls(t)
        for c in getColumns(t) do
            if c ∉ getColumns(nns) then
                mutant ← createotNull(getColumns(nns) ∪ {c})
                mutants ← mutants ∪ {createMutant(nns,mutant)}
    return mutants
```

NNA: The function in Algorithm 10 describes the approach of the `NOT NULL` constraint addition (NNA) operator. This generates mutants by adding a `NOT NULL` constraint to each column of the schema in turn, unless the column already has one defined on it. So, given the table `x`:

```
x (a INT NOT NULL, b INT, c INT)
```

...the mutation operator is applied to columns `b` and `c` in turn, but not `a` as it already has a `NOT NULL` constraint, to give the mutants m_1 and m_2:

```
m1:  x (a INT NOT NULL, b INT NOT NULL, c INT)

m2:  x (a INT NOT NULL, b INT, c INT NOT NULL)
```

The addition of these constraints reduces the data that can be accepted into the database, with the mutants modelling the omission of a `NOT NULL` constraint.

Algorithm 11 Not null removal operator

```
function NNCR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        nns ← getNotNulls(t)
        for nn in nns do
            mutant ← nns \ {nn}
            mutants ← mutants ∪ {createMutant(nns,mutant)}
    return mutants
```

NNR: The function in Algorithm 11 describes the approach of the `NOT NULL` constraint removal (NNR) operator. This operator removes existing `NOT NULL` constraints, one at a time, from a schema to produce mutants. Therefore, applying this operator to a table `x`:

```
x (a INT NOT NULL, b INT NOT NULL, c INT)
```

...will yield two mutants, $\texttt{m}_1$ and $\texttt{m}_2$, with the constraints on `a` and `b` removed one at a time:

$\texttt{m}_1$: `x (a INT, b INT NOT NULL, c INT)`

$\texttt{m}_2$: `x (a INT NOT NULL, b INT, c INT)`

These mutants model the programmer mistake of erroneously including a `NOT NULL` constraint, thus preventing the accepting of a `NULL` value into the database.

4.3.5 `CHECK` constraint operators

Algorithm 12 Check constraint removal operator

```
function CCR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        checks ← getChecks(t)
        for c in checks do
            mutantChecks ← checks \ {c}
            mutants ← mutants ∪ {createMutant(checks,mutantChecks)}
    return mutants
```

CR: The function in Algorithm 12 describes the approach of the `CHECK` constraint removal (CR) operator. Provided that the schema contains at least one `CHECK` constraint, this produces mutants by removing each `CHECK` constraint from a schema, one at a time. It does not remove individual branches of a complex `CHECK` expression, so cannot model mistakes that may have been made inside the expression itself. For example, given the table `x`:

```
x (a INT, b INT, CHECK (a < b), CHECK (a > 0 AND b > 0))
```

...the operator will produce the two mutants, $\texttt{m}_1$ and $\texttt{m}_2$, by fully removing each `CHECK` constraint:

```
m1:  x (a INT, b INT, CHECK (a > 0 AND b > 0))

m2:  x (a INT, b INT, CHECK (a < b))
```

These mutants each model the programmer error of including an unneeded `CHECK` constraint, thus over-constraining the data that can be accepted into the database.

Algorithm 13 Check constraint IN LIST element removal operator

```
function CCInListelementR(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for ch in getChecks(t) do
            for expr in getExpressions(ch) do
                if containsInListExpr(expr) then
                    mutants ← mutants ∪ MutateList(ch, expr)
    return mutants
function MutateList(expr)
    mutants ← ∅
    for i in getExpressionList(expr) do
        mutants ← mutants ∪ mutate(expr, getExpressionList(expr) \ {i})
    return mutants
```

CInListElementR: The function in Algorithm 13 describes the approach of the `CHECK` constraint `IN` list element removal (CInListElementR) operator. Mutants are produced with this operator by removing a single element from the list in the right-hand side of the `CHECK` constraint expression ***expr*** `IN (`$\texttt{x}_1$`, `$\texttt{x}_2$`, ..., `$\texttt{x}_n$`)`. For example, given the constraint:

```
CHECK (expr IN (x,y,z))
```

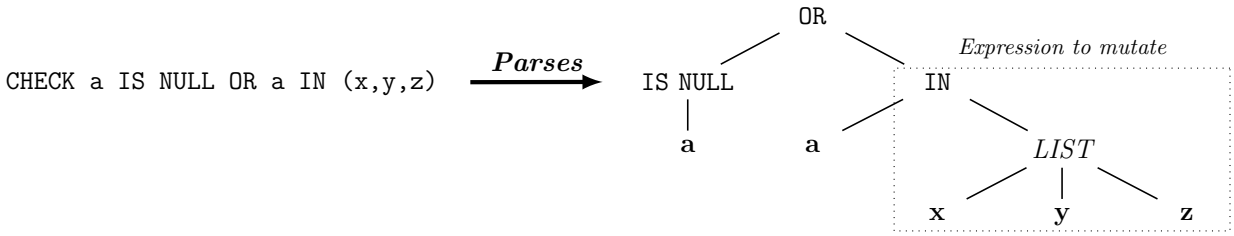
...this operator would produce three mutants — $\texttt{m}_1$, $\texttt{m}_2$ and $\texttt{m}_3$ — by removing the elements ***x***, ***y*** and ***z*** from the right-hand side of the expression in turn:

```
m1:  CHECK (expr IN (y,z))

m2:  CHECK (expr IN (x,z))

m3:  CHECK (expr IN (x,y))
```

In this example, the variables ***x***, ***y*** and ***z*** may be column references, constant values or sub-expressions. The operator may also be applied within complex `CHECK` constraints where expressions are nested, for example with the expression:



By traversing the expression from the "root" node – in this case `OR` – the mutation operator can locate any `IN` expression that it can mutate – `IN (x,y,z)` for this example. Then mutation can proceed as previously described. The mutants produced with this operator model the mistake of a programmer erroneously including an extra value in the right-hand side of a `IN` expression in a `CHECK` constraint.

Algorithm 14 Check constraint relational operation exchange operator

```
function CRelOpE(schema)
    mutants ← ∅
    for t in getTables(schema) do
        for ch in getChecks(t) do
            for expr in getExpressions(ch) do
                if containsRelOpExpr(expr) then
                    mutants ← mutants ∪ MutateRelOp(expr)
    return mutants

function MutateRelOp(expr)
    mutants ← ∅
    for op in {=, ≠, <, >, ≤, ≥} do
        if op ≠ getExpressionRelOp(expr) then
            mutants ← mutants ∪ {mutate(getExpressionRelOp(expr), op)}
    return mutants
```

CRelOpE: The function in Algorithm 14 describes the approach of the `CHECK` constraint relational operator exchange (CRelOpE) operator. To generate mutants, this operator exchanges the relational operator in a relational expression (such as `a > 5`) with another relational operator ($=, \neq, <, \leq, \geq$). For example, using the table `x`:

```
x (a INT, CHECK a > 5)
```

...there are five mutants generated — one for each relational operator other than the original — shown below as m_1 to m_5:

m_1: `x (a INT, CHECK a = 5)`

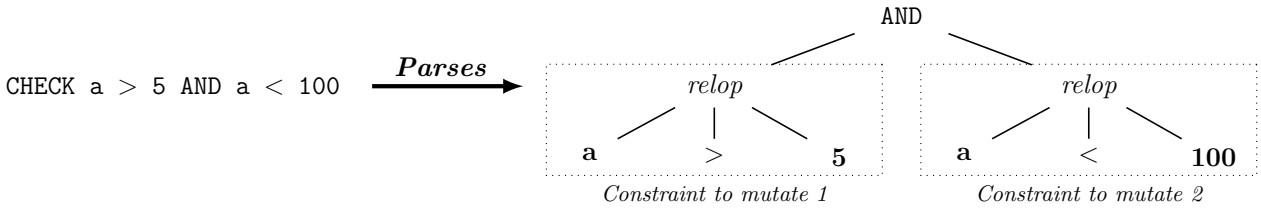
m_2: `x (a INT, CHECK a` $\neq$ `5)`

m_3: `x (a INT, CHECK a` $<$ `5)`

m_4: `x (a INT, CHECK a` $\leq$ `5)`

m_5: `x (a INT, CHECK a` $\geq$ `5)`

The operator can be applied to any `CHECK` expression with a relational expression in the form ***x relop y***, where ***x*** and ***y*** may be a column reference, constant value or sub-expression, and ***relop*** is a relational operator. In addition, the relational expression may be nested inside a more complex expression, such as `CHECK a IS NULL OR (a > 5)`, using the expression tree traversal described for CInListElementR. Where a single `CHECK` constraint contains more than one relational expression, mutants will be produced for each independently, such that only one constraint is mutated at a time. For example, in the expression `CHECK a > 5 AND a < 100` the mutation operator would be applied twice to the expression tree:



Applying the operator independently to these two points would produce a total of 10 mutants, which could divided into two sets — the first where `a > 5` has been mutated and `a < 100` kept constant and the second the where `a < 100` has been mutated and `a > 5` kept constant, show below as m_1 to m_5 and m_6 to m_{10}, respectively:

Mutants of constraint 1:		*Mutants of constraint 2:*	
m_1:	`CHECK a = 5 AND a < 100`	m_6:	`CHECK a > 5 AND a = 100`
m_2:	`CHECK a ≠ 5 AND a < 100`	m_7:	`CHECK a > 5 AND a ≠ 100`
m_3:	`CHECK a < 5 AND a < 100`	m_8:	`CHECK a > 5 AND a > 100`
m_4:	`CHECK a ≤ 5 AND a < 100`	m_9:	`CHECK a > 5 AND a ≤ 100`
m_5:	`CHECK a ≥ 5 AND a < 100`	m_{10}:	`CHECK a > 5 AND a ≥ 100`

4.4 Operator Productivity

While the number of mutants produced for each of the mutation operators described in Section 4.3 have been analysed previously [118], their productivity has not been defined in terms of the elements of the schema being mutated before this point. This Section provides those definitions, described in terms of the mutation utility functions in Figure 4.1. In each case, the productivity is specified with respect to a single table, *tbl*.

Calculating the productivity of each operator without needing to execute them against a schema to generate mutants, which may prove costly for very large schemas, is useful for a number of different use cases, such as:

Selective mutation This technique aims to reduce the number of mutants that need to be analysed by applying rules such as *"do not apply the N most productive operators"*. Determining how much the cost of executing mutation analysis changes with varying values for N could be performed entirely statically using these definitions, reducing the cost of such experimentation. Also, as these definitions are given with respect to each table of a schema, the operators omitted can be easily varied for each different schema under test.

Schema migration When making changes to the schema, it is generally unknown what scope there may be for the changes to introduce new faults. However, using the definitions of operator productivity it could be possible to estimate the potential increase in scope a particular modification to the schema may introduce. This can

be calculated by finding the difference between the number of mutants that would be produced for a schema before and after the change. In the worst case, each additional mutant represents an additional test case that is required to ensure that a fault has not been introduced by the schema migration.

Operator introduction A user of the mutation analysis techniques described in this thesis might wish to reduce the cost of evaluating their test suite by selecting a subset of mutation operator that represent faults they most wish to detect. However, in deciding when it might be useful to include additional operators to that set it is important for them to estimate the likely change in cost of the analysis. The definitions in this Section could be used to support such decision making by providing a static technique to approximate the mutant-cost of including extra operators.

Throughout this Section, each definition is applied to the following `stock` table, which could be used to store basic information about a set of products a business has, which is defined as follows[1]:

```
CREATE TABLE stock (
    id integer PRIMARY KEY,
    name varchar(50) UNIQUE,
    description varchar(50),
    type varchar(10),
    price numeric NOT NULL,
    location varchar(10) REFERENCES
        store(name),
    CHECK (price > 0),
    CHECK type IN ('mens','womens','
        childrens')
);
```

4.4.1 `PRIMARY KEY` operators

PKColumnA: The `PRIMARY KEY` constraint column addition operator creates a mutant by adding a column to the `PRIMARY KEY` of the table, provided the column is not already included in the constraint. The productivity can therefore be defined as:

[1]For clarity, the `store` table has been omitted as the operator productivity definitions are given per table. For the `FOREIGN KEY` operators assumptions for the calculations will be stated where necessary.

$$\|\textsc{getColumns}(tbl)\| - \|\textsc{getColumns}(\textsc{getPrimaryKey}(tbl))\|$$

By applying this to the `stock` table, the number of mutants produced by this operator could be calculated as follows:

$$\|\{id, name, description, type, price, location\}\| - \|\{id\}\| = 5$$

PKColumnR: The `PRIMARY KEY` constraint column removal operator produces each mutant by removing a column from the `PRIMARY KEY` of a table in turn, provided one is defined in the schema. The operator productivity is therefore simply:

$$\|\textsc{getColumns}(\textsc{getPrimaryKey}(tbl))\|$$

When applying this definition to the `stock` example schema, the number of mutants produced can be calculated easily:

$$\|\{id\}\| = 1$$

PKColumnE: The `PRIMARY KEY` constraint column exchange operator replaces each column in a `PRIMARY KEY` with each column not already included in the constraint. Consequently, the productivity of the operator can be expressed as:

$$(\|\textsc{getColumns}(tbl)\| - \|\textsc{getColumns}(\textsc{getPrimaryKey}(tbl))\|) \cdot \|\textsc{getColumns}(\textsc{getPrimaryKey}(tbl))\|$$

Using this definition it is possible to calculate the number of mutants that would be generated by the operator for the `stock` schema as follows:

$$(\|\{id, name, description, type, price, location\}\| - \|\{id\}\|) \cdot \|\{id\}\| = (6-1) \cdot 1 = 5$$

4.4.2 `FOREIGN KEY` operators

FKColumnPairA: The `FOREIGN KEY` column pair addition operator adds new pairs of columns (one each from the local and reference tables) to existing `FOREIGN KEY` constraints to produce mutants. These pairs must be matched in data type, to ensure the mutants are valid. The productivity of this operator can be defined as:

$$\sum_{i=1}^{\|\textsc{getForeignKeys}(\text{tbl})\|} \left\| \left\{ a, b \mid a \in \textsc{getColumns}(tbl), \right. \right.$$
$$b \in \textsc{getColumns}(\textsc{getRefTable}(\textsc{getForeignKey}(tbl, i))),$$
$$\textsc{getType}(a) = \textsc{getType}(b),$$
$$\left. \left. (a, b) \notin \textsc{getForeignKeyPairs}(\textsc{getForeignKey}(tbl, i)) \right\} \right\|$$

This definition can be applied to the `stock` schema as follows. Assume that the `store` table contains two columns, defined as `id integer` and `name varchar(50)`. This gives 6 columns in `stock` and 2 in `store`, yielding an initial set of 12 column pairings. Of these, the type-matching restriction means that `stock.id` can only be matched with `store.id`, while `store.name` can be matched with `stock.name`, `stock.description`, `stock.type` and `stock.location`. Of these, the pairing of `stock.location` and `stores.name` is already part of the `FOREIGN KEY` so is removed from the pool of potential mutants. This leaves a total of 5 mutants that would be produced by applying this operator.

FKColumnPairR: The `FOREIGN KEY` column pair removal operator generates mutants by discarding each pair of columns, one from each of the 'local' and 'reference' tables, from a `FOREIGN KEY` constraint in turn, repeating the process for each `FOREIGN KEY` in a table. This gives the following overall productivity:

$$\sum_{i=1}^{\|\textsc{getForeignKeys}(tbl)\|} \|\textsc{getForeignKeyPairs}(\textsc{getForeignKey}(tbl, i))\|$$

Using this definition with the `stock` schema states that only one mutant would be produced by this operator, as there is only one `FOREIGN KEY` to be removed and it contains only a single local-reference column pair.

FKColumnPairE: The `FOREIGN KEY` column pair exchange operator generates mutants by exchanging pairs of columns in a `FOREIGN KEY` constraint with columns not already in the constraint. To guarantee the constraint remains valid the data types of these columns must match. Overall, the productivity of this operator can be described with the same expression as for the FKColumnPairA operator:

$$\sum_{i=1}^{\|\textsc{getForeignKeys}(\text{tbl})\|} \left\| \Big\{ a, b \mid a \in \textsc{getColumns}(tbl), \; b \in \textsc{getColumns}(\textsc{getRefTable}(\textsc{getForeignKey}(tbl, i))), \; \textsc{getType}(a) = \textsc{getType}(b), \; (a, b) \notin \textsc{getForeignKeyPairs}(\textsc{getForeignKey}(tbl, i)) \Big\} \right\|$$

The application of this definition to the `stock` example schema is the same as for the *FKColumnPairA* operator, so yields the same total of 5 mutants, applying the same assumptions discussed above.

4.4.3 `UNIQUE` constraint operators

UCColumnA: The `UNIQUE` constraint column addition operator creates mutants by firstly adding new `UNIQUE` constraints for each column that is not subject to a single-column `UNIQUE` constraint, and secondly adding each column to existing `UNIQUE` constraints that is not already included in. These two phases are represented as the first and second parenthesised parts of the following productivity expression:

$$\left(\|\{c \in \textsc{getColumns}(tbl) \mid \textsc{getUnique}(c) \notin \textsc{getUniques}(tbl)\}\| \right) + \left(\sum_{i=1}^{\|\textsc{getUniques}(\text{tbl})\|} \|\textsc{getColumns}(tbl)\| \; - \; \|\textsc{getColumns}(\textsc{getUnique}(tbl, i))\| \right)$$

This can be applied to the `stock` table to calculate the number of mutants that would be generated as follows (in this case both phases are effectively the same because the `FOREIGN KEY` contains a single column pair):

$$(\ \|\{id, description, type, price, location\}\|\ ) + (\ \|\{id, description, type, price, location\}\|\ ) = 10$$

UCColumnR: The `UNIQUE` constraint column removal operator takes one column out of a `UNIQUE` constraint at a time to produce mutants. This is then repeated for each `UNIQUE` constraint in the table. Overall, the productivity of this operator is expressible as:

$$\sum_{i=1}^{\|\textsc{getUniques}(tbl)\|} \|\textsc{getColumns}(\textsc{getUnique}(tbl, i))\|$$

Intuitively, because there is only one `UNIQUE` constraint in the `stock` table, applying it to this schema simplifies the expression such that the number of mutants is exactly the number of columns in that constraint – so, in this case it would determine that only one mutant would be produced by this operator were it applied.

UCColumnE: The `UNIQUE` constraint column exchange operator generates mutants of each `UNIQUE` constraint by swapping every column in the constraint with columns not already included in that constraint. This gives the following productivity:

$$\sum_{i=1}^{\|\textsc{getUniques}(tbl)\|} (\|\textsc{getColumns}(tbl)\| - \|\textsc{getColumns}(\textsc{getUnique}(tbl, i))\|) \cdot \|\textsc{getColumns}(\textsc{getUnique}(tbl, i))\|$$

Using this productivity definition with the `stock` table gives the following equation, showing a total of 5 mutants would be generated by applying the operator to it:

$$(\|\{id, name, description, type, price, location\}\| - \|\{name\}\|) \cdot \|\{name\}\| = (6-1) \cdot 1 = 5$$

4.4.4 `NOT NULL` constraint operators

NNA: The `NOT NULL` addition operator mutates tables by adding a `NOT NULL` constraint to each column in turn, provided that column isn't already subject to one. This gives a simple productivity expression of:

$$\|\textsc{getColumns}(tbl)\| - \|\textsc{getNotNulls}(tbl)\|$$

As the `stock` table contains 6 columns with 1 already subject to a `NOT NULL` constraint, this definition reveals a total of 5 mutants would be generated if this operator were executed with it.

NNR: The `NOT NULL` removal operator produces mutants by discarding each `NOT NULL` constraint defined on a table in turn. The productivity of this operator can be described as follows:

$$\|\textsc{getNotNulls}(tbl)\|$$

Applying this expression with the `stock` table shows that only one mutant would be produced, as it contains only a single `NOT NULL` constraint on the `price` column.

4.4.5 `CHECK` constraint operators

CR: The `CHECK` constraint removal operator mutates a table by taking out each `CHECK` constraint, in its entirety. This gives a productivity of:

$$\|\textsc{getChecks}(tbl)\|$$

Because the `stock` table contains one `CHECK` constraint, applying this definition would identify that a single mutant would be produced by this operator.

CInListElementR: The `CHECK` constraint `IN` list element removal operator produces mutants of expressions in `CHECK` constraints with the form ***expr*** `IN (`$\texttt{x}_1$`, `$\texttt{x}_2$`, ..., `$\texttt{x}_n$`)` by removing each of the list elements, $\texttt{x}_i$, in turn. The productivity of this operator can be described as:

$$\sum_{i=1}^{\|\textsc{getChecks}(tbl)\|} \|\textsc{getExpressionList}(\textsc{getCheck}(tbl, i))\|$$

As shown in the definition for the `stock` table, the `type` column is restricted to one of three fixed values. Therefore, applying the above definition to this table would show a total of three mutants would be produced.

CRelOpE: The `CHECK` constraint relational operator exchange operator produces mutants of `CHECK` constraints that contain relational expressions (such as `a` > 5) by replacing the existing relational operator with each alternative operator ($=,\neq,<,\leq,\geq$). Therefore, the operator productivity can be expressed as:

$$\sum_{i=1}^{\|\text{getChecks}(tbl)\|} \|\text{getExpressionRelOp}(\text{getCheck}(tbl, i))\| \cdot 5$$

Given that the `stock` table contains a single `CHECK` constraint with relational expression, evaluating this expression shows that 5 mutants would be produced if the operator was applied to the schema.

4.5 Alternative Elements to Mutate

While the mutation operators defined in this Chapter are able to model a wide range of possible faults in the integrity constraints of a relational database schema, including both faults of omission and commission, there are a number of alternative elements of a schema that could be mutated. The remainder of this Section discusses a range of these, along with discussion of why operators for mutating these were not included in the initial collection of mutation operators proposed in this thesis.

4.5.1 Column definitions

As previously described, even in relational database schemas with no integrity constraints there are still restrictions on the data that can be stored in a table according to the data types and lengths specified for each column. These are referred to as domain constraints [100], and must be specified as part of any relational schema expressed in the SQL language. For example, consider the following schema:

```
CREATE TABLE stock (
    product_id integer,
    description varchar(50),
    price numeric,
    sale_price numeric
);
```

In this case, `product_id` must be a whole number, `description` can only store a character string with a maximum length of 50, and both `price` and `sale_price` must be a floating point number. As with the integrity constraints of the schema, mistakes made when defining the domain constraints of a schema may cause difficult to detect bugs within applications that use the database. For example, using a floating point column type rather than an integer type may lead to `CHECK` constraints that test if the column is equal to a constant value to fail as floating point types do not store values exactly, instead storing a very close approximation of the number. Alternatively, incorrectly specifying the length a column may lead to values being truncated, if the given length is too small, leading to inaccuracies in the stored data. This problem may also be exacerbated by subtle differences between the handling of certain data types by different DBMSs.

While mutation operators could be proposed to mutate both the type of columns, for example replacing integer and floating point types, or altering the length specified for columns, increasing or decreasing it, this could lead to a very large number of mutants being produced to analyse. In addition, killing each of these mutants would require very specific test cases to be produced, leading to large test suites that may prove infeasible to run frequently in a practical setting. Therefore, these mutation operators were not considered as part of this thesis.

4.5.2 Further `CHECK` mutation

Although three mutation operators were defined for modifying `CHECK` constraints earlier in this Chapter – **CR**, **CInListElementR** and **CRelOpE** – there are many additional mutation operators that could be proposed, taking inspiration from operators for other programming languages. For example, considering the following `CHECK` constraint:

```
CHECK ((x != 5) AND (y IS NOT NULL))
```

...the following additional mutations could be considered:

1. Modifying the literal value 5. This could be by incrementing, decrementing, negating, or setting it to zero. Alternatively, it could be replaced with a column reference.

2. Swapping the column references `x` and `y` with each other valid column reference, negating them or setting them to a constant value.

3. Changing `IS NOT NULL` to `IS NULL`.

4. Replacing the logical operator `AND` with `OR`.

5. Negating each of the two clauses in turn using the `NOT` operator.

6. Removing each conditional clause in turn.

These operators could be defined using the mutation framework developed as part of this thesis, however were not included in the operators considered. This is because `CHECK` constraints seem to be a relatively infrequently used type of constraint in relational schemas, as demonstrated by the attributes of the schemas available in SchemaAnalyst, shown in Table 3.2 (see page 92). These schemas were gathered from a wide range of sources, including real-world applications, so can be considered representative of a large number of schemas used in database applications. Because there are very few `CHECK` constraints in these schemas it would be difficult to evaluate a set of operators implementing the above mutations as they would produce very few mutants, and in some cases no mutants for any schema. However, if additional schemas containing more examples of `CHECK` constraints were acquired these operators could be considered as part of future work.

4.5.3 Table indices

When defining a relational database schema the programmer can optionally include a number of `INDEX` statements. Each of these defines an index that the DBMS should maintain that can be used to more quickly lookup values for one or more columns, at the expense of increases the time taken for each `INSERT` statement [100]. For example, considering the schema:

```
CREATE TABLE users (id int, name varchar(50), address varchar(50));
CREATE INDEX users_index ON users (id);
```

In this case, querying the database with a `WHERE` clause that searches the `id` column, such as `WHERE id == 5`, will be faster than if the index `users_index` had not been defined. If the table were to grow very large and was subject to many more read operations

than write operations, then this could improve performance of the database significantly. However, it is important to note that the presence of an index does not alter the results of a query – that is, the difference in behaviour is entirely related to performance of the database[2].

Considering the example schema, there are a number of possible mutations that could be applied:

1. Add each other column reference to the existing index.
2. Replace the column reference `id` with each other column.
3. Add an index to each other column.
4. Remove the existing index.

Although the mutation framework in SchemaAnalyst could be used to implement mutation operators for these modifications each of these would only affect the performance of the database, and not its overall behaviour. This is in contrast to the operators defined earlier in this Chapter that focus of programmer mistakes that alter the behaviour of the database, which may in turn lead to errors in applications using the database. Therefore mutation operators for `INDEX` statements were not considered as part of this thesis, although could be investigated as an item of future work.

4.6 Summary

This Chapter described a set of 14 mutation operators that can be applied each of the major kinds of SQL constraint – `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `NOT NULL` and `CHECK` constraints. A naming scheme for these operators was specified, which gives each a unique name by combining the kind of constraint with the type of modification made – addition, removal and exchange. Algorithms were described for each of the operators,

[2]An exception to this is a unique index, which behaves exactly as a `UNIQUE` constraint. In SchemaAnalyst this type of index is instead converted to a `UNIQUE` constraint which is modified using the mutation operators defined in this thesis.

which represent a contribution of this thesis. In addition, the productivity of each operator was defined, in terms of the attributes of the schema under mutation. Finally, alternative elements of a schema that could be mutated were described and mutations suggested, along with brief discussion of why these were omitted from this thesis. These operators are now used in Chapters 5 to 7 as part of mutation analysis experiments that investigate various facets of mutation analysis of relational database schemas.

Chapter 5

Evaluating Coverage Criteria for Relational Database Schemas Using Mutation Analysis

5.1 Introduction

Making use of the mutation framework detailed in Section 3.6, this Chapter combines the algorithm described in Section 1.3.2 and mutation operators defined in Chapter 4 to perform a mutation analysis experiment to evaluate the data generation component of the SchemaAnalyst tool. By modelling a range of faults in integrity constraints, this enables the different coverage criteria and search technique combinations, discussed previously in Section 3.5, to be compared in terms of their capability to detect realistic faults. As a result, those configurations that would be most effective for testing schemas in an industrial setting can be identified, such that the widest range of faults can be detected.

To ensure the result were generalisable and to facilitate to analysis of how different configuration options may impact upon effectiveness, the empirical experiment describe in this Chapter includes 32 schemas, all nine of the coverage criteria discussed in Section 3.5.1, both available search algorithms and all three supported DBMSs. The results revealed that the choice of search algorithm affects the fault-finding capability of some

coverage criteria, and also that combining test cases from different types of criteria can produce more effective test suites.

The contributions this Chapter makes are:

1. A description of an empirical experiment designed to determine the effectiveness of the search techniques, in terms of producing data to test all required constraints, and of the coverage criteria, in terms of mutant killing ability; and

2. Discussion of the results of the empirical experiment in the context of four research questions, including how combining criteria can improve the mutant killing ability of a test suite.

5.2 Experiment Design

5.2.1 Schemas

To ensure that the complexity of the schemas (e.g., number of tables and constraints) does not adversely bias the results of this experiment, a total of 32 schemas were chosen that range in complexity, as shown in Table 5.1. As previously discussed in Section 3.4, these schemas were acquired from a variety of sources, including real-world applications. For example, RiskIt, taken from an insurance risk modelling application, and MozillaPermissions, used in the popular Firefox Internet browser. In addition, care was taken to ensure the schemas contained examples of each major constraint type (i.e., `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `CHECK` and `NOT NULL`), as well as single-column and multi-column variants where appropriate (e.g., `PRIMARY KEY (x)` and `PRIMARY KEY (x,y)`). The varying complexity and source of the schemas used ensures that the results are representative of a wide range of schemas, and therefore that any conclusions can be generalised to other schemas. Meanwhile, the selection of 32 schemas from those available in SchemaAnalyst ensures the execution cost of the experiment is not prohibitive.

Table 5.1: Schemas analysed in the empirical study

Schema	Tables	Columns	∑Constraints	Checks	Foreign Keys	Not Nulls	Primary Keys	Uniques
ArtistSimilarity	2	3	3	0	2	0	1	0
ArtistTerm	5	7	7	0	4	0	3	0
BankAccount	2	9	8	0	1	5	2	0
BookTown	22	67	28	2	0	15	11	0
BrowserCookies	2	13	10	2	1	4	2	1
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	19	0	4	10	5	0
CustomerOrder	7	32	42	1	7	27	7	0
DellStore	8	52	39	0	0	39	0	0
Employee	1	7	4	3	0	0	1	0
Examination	2	21	9	6	1	0	2	0
Flights	2	13	10	1	1	6	2	0
FrenchTowns	3	14	24	0	2	13	0	9
Inventory	1	4	2	0	0	0	1	1
Iso3166	1	3	3	0	0	2	1	0
iTrust	42	309	134	8	1	88	37	0
JWhoisServer	6	49	50	0	0	44	6	0
MozillaExtensions	6	51	7	0	0	0	2	5
MozillaPermissions	1	8	1	0	0	0	1	0
NistDML181	2	7	2	0	1	0	1	0
NistDML182	2	32	2	0	1	0	1	0
NistDML183	2	6	2	0	1	0	0	1
NistWeather	2	9	13	5	1	5	2	0
NistXTS748	1	3	3	1	0	1	0	1
NistXTS749	2	7	7	1	1	3	2	0
Person	1	5	7	1	0	5	1	0
Products	3	9	14	4	2	5	3	0
RiskIt	13	57	36	0	10	15	11	0
StackOverflow	4	43	5	0	0	5	0	0
StudentResidence	2	6	8	3	1	2	2	0
UnixUsage	8	32	24	0	7	10	7	0
Usda	10	67	31	0	0	31	0	0
Total	172	975	554	38	49	335	114	18

5.2.2 DBMSs

As DBMSs may vary slightly in their interpretation of the SQL specification, the data generation component of SchemaAnalyst produces subtly different test data depending on the DBMS in use. Therefore it is important to determine whether the effectiveness of a given criterion and search algorithm is consistent across different DBMSs. For that reason, this experiment includes all three DBMSs supported by SchemaAnalyst — PostgreSQL, HyperSQL and SQLite. The differences in architecture and design philosophy between these DBMSs was discussed briefly in Section 3.3.3.

5.2.3 Methodology

Given the complexity of the data generation component in SchemaAnalyst, such as the number of different coverage criteria and search algorithms (described in Sections 3.5.1 and 3.5.2 respectively), this empirical experiment is divided into a number of parts that analyse a number of different facets.

Firstly, the nine coverage criteria are compared in terms of the number of test cases they require, across all of the 32 schemas, and how frequently the two search algorithms are able to successfully produce data for those test cases. In this case the combination is possibly important as a more complex coverage criterion may require a more guided search algorithm to produce data that satisfies the predicates and therefore produces usable test cases.

Secondly, mutation analysis is used to compare the test cases generated by each of the coverage criteria and search algorithm pairings in terms of fault-finding capability. The higher proportion of mutants killed by the test cases of a given criterion, the more likely that those test cases would reveal faults in a real-world application, and therefore the more useful that criterion is for generating test data.

Next, data gathered from the mutation analysis trials is used to determine which mutants were killed by specific coverage criterion, to examine whether some criteria are better suited to exposing certain types of faults. This is categorised by the operators used to produce the mutants and what proportion of those mutants were killed.

Finally, the effectiveness of combining coverage criteria is analysed, using the data detailing which mutants have been killed by each criterion, to identify whether combining test cases from the different classes of criterion (i.e., constraint criteria, unique column criteria and null column criteria) can yield a test suite with a higher mutation score. If so, this would suggest that such a combined test suite would be the most effective at revealing faults in a real-world application.

In summary, the experiment involves executing nine coverage criteria (five constraint coverage, two unique coverage and two null column coverage) to generate the predicates for 32 schemas, using three different DBMSs. The number of predicates is recorded in every case, with 30 repeat trials with different random seeds to reduce the impact of

the stochastic nature of the search techniques and allow averages to be obtained. This requires a total of 25,920 experimental trials. Where the created predicate is detected to be trivially infeasible[1] during the data generation processed (e.g., specifying $x = \texttt{NULL} \wedge x \neq \texttt{NULL}$) or duplicated[2] (i.e., two test cases have the same predicates) this is also recorded. For each set of test predicates produced, two search algorithms are used to attempt to generate data that satisfies them and therefore produce test cases, evaluating the algorithms based on their success rate. The effectiveness of those test cases is calculated as the mutation score obtained from mutation analysis, using the full set of 14 operators specified in Chapter 4. Using data from the mutation analysis a "per-operator" mutation score is produced to identify whether certain criteria kill mutants produced by particular operators more frequently. The mutation score is then calculated for each permutation of constraint criteria, unique column criteria and null column criteria, to determine the effectiveness of combined test suites, and whether such combinations can produce tests that identify a higher proportion of faults.

5.2.4 Configuration

Because of the large number of experimental trials, described in the previous Section, this experiment was executed using a high performance computing cluster, running 64-bit Scientific Linux and managed with Sun Grid Engine. The SchemaAnalyst tool was compiled using Java Development Kit 7 and executed with the 64-bit Oracle Java 1.7 virtual machine. The experiment used PostgreSQL 9.1.9 in its default configuration, and HyperSQL version 2.28 and SQLite 3.6.20 using their "in-memory" settings. The data generation algorithm was configured to allow up to 100,000 fitness evaluations, which was shown in initial trials to be sufficient to successfully generate data whilst avoiding excessive execution time in the cases where the search process is highly unlikely to find data that satisfies the set of data generation constraints.

[1] Currently, only two cases of infeasibility are detected using static analysis of the test predicate (see [76] for more detail), with both relating to requiring a variable to be both `NULL` and $\neg$`NULL` simulatenously. In the first case, the test predicate requires $a = \texttt{NULL} \wedge a = \neg\texttt{NULL}$. In the second case, this is extended to allow for a logical `OR` nested within an `AND` where all subclauses of the former are negations of the latter, for example:

$$a = \texttt{NULL} \wedge b = \texttt{NULL} \wedge (a = \neg\texttt{NULL} \vee b = \neg\texttt{NULL})$$

[2] The removal of duplicate requirements is achieved using a `Set`-based implementation whereby only one of two duplicate requirements will be stored when joining multiple requirements together.

5.2.5 Threats to validity

As with all empirical experiments, there are a number of different threats to the validity of the results. This Section describes some of the most significant of these and explains how their impact was mitigated to ensure the findings are both accurate and generalisable.

DBMS specific results The differences in behaviour between DBMSs may lead to varied results according to which is being used – for example, certain coverage criteria may be more or less effective depending upon how `NULL` values are handled. The impact of this is reduced by including three different DBMSs, including SQLite which is more permissive with respect to `NULL` values.

Stochasticity of algorithms As the data generation techniques use randomly generated values to search for valid test data, the performance of a particular configuration may vary significantly due to the seed value. To avoid this affecting the empirical results, 30 repeat trials were performed for each condition with a different random seed for each, to allow averages to be calculated.

Bias from search algorithm The effectiveness of the data produced for a particular coverage criterion may be affected by the search algorithm used to generate the data. This threat was managed by including two different data generation algorithms with significantly different designs.

5.3 Empirical Results

The results of the experiment described in this Chapter are now discussed in the context of four research questions, each evaluating different facets of the SchemaAnalyst data generation component.

RQ1: *How does the choice of coverage criterion affect the number of test cases required for each schema, how successful are the two search algorithms at producing data for those tests, and are these results dependant on the DBMS used?*

Table 5.2: Number of test cases generated for each schema by the coverage criteria.

For each criterion, the left value is for PostgreSQL and HyperSQL (which have identical results due to their similarity in handling of constraints) and the right value for SQLite. The values in brackets are the number of test cases once trivially infeasible and duplicate cases have been removed.

(a) Constraint coverage criteria

Schema	APC		ICC		AICC		CondAICC		ClauseAICC	
ArtistSimilarity	4 (4)	4 (4)	6 (6)	6 (6)	6 (5)	6 (5)	9 (9)	9 (9)	9 (9)	9 (9)
ArtistTerm	10 (10)	10 (10)	14 (14)	14 (14)	14 (12)	14 (12)	21 (21)	21 (21)	21 (21)	21 (21)
BankAccount	4 (4)	4 (4)	16 (16)	16 (16)	16 (10)	16 (10)	19 (13)	19 (15)	19 (13)	19 (15)
BookTown	26 (26)	26 (26)	34 (34)	56 (56)	34 (30)	56 (41)	47 (42)	69 (53)	49 (44)	71 (55)
BrowserCookies	4 (4)	4 (4)	14 (14)	20 (20)	14 (9)	20 (12)	20 (19)	26 (22)	30 (27)	36 (29)
Cloc	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
CoffeeOrders	10 (10)	10 (10)	38 (38)	38 (38)	38 (24)	38 (24)	47 (37)	47 (40)	47 (37)	47 (40)
CustomerOrder	14 (14)	14 (14)	70 (70)	84 (84)	70 (42)	84 (49)	85 (58)	99 (65)	86 (59)	100 (66)
DellStore	16 (16)	16 (16)	78 (78)	78 (78)	78 (47)	78 (47)	78 (47)	78 (47)	78 (47)	78 (47)
Employee	2 (2)	2 (2)	8 (8)	8 (8)	8 (5)	8 (5)	12 (11)	12 (12)	12 (11)	12 (12)
Examination	4 (4)	4 (4)	18 (18)	18 (18)	18 (11)	18 (11)	27 (27)	27 (27)	27 (27)	27 (27)
Flights	4 (4)	4 (4)	12 (12)	20 (20)	12 (8)	20 (12)	16 (12)	24 (16)	25 (22)	33 (22)
FrenchTowns	6 (6)	6 (6)	48 (48)	48 (48)	48 (27)	48 (27)	59 (38)	59 (38)	61 (39)	61 (39)
Inventory	2 (2)	2 (2)	4 (4)	4 (4)	4 (3)	4 (3)	6 (6)	6 (6)	6 (6)	6 (6)
Iso3166	2 (2)	2 (2)	6 (6)	6 (6)	6 (4)	6 (4)	7 (5)	7 (6)	7 (5)	7 (6)
iTrust	82 (82)	82 (82)	234 (234)	268 (268)	234 (158)	268 (175)	280 (208)	314 (243)	559 (488)	593 (517)
JWhoisServer	12 (12)	12 (12)	88 (88)	100 (100)	88 (50)	100 (56)	94 (56)	106 (62)	94 (56)	106 (62)
MozillaExtensions	10 (10)	10 (10)	14 (14)	14 (14)	14 (12)	14 (12)	21 (21)	21 (21)	31 (31)	31 (31)
MozillaPermissions	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	3 (3)	3 (3)	3 (3)	3 (3)
NistDML181	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	6 (6)	6 (6)	10 (10)	10 (10)
NistDML182	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	6 (6)	6 (6)	62 (62)	62 (62)
NistDML183	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	4 (4)	6 (6)	6 (6)	10 (10)	10 (10)
NistWeather	4 (4)	4 (4)	24 (24)	26 (26)	24 (14)	26 (15)	32 (22)	34 (26)	39 (30)	41 (32)
NistXTS748	2 (2)	2 (2)	6 (6)	6 (6)	6 (4)	6 (4)	8 (8)	8 (8)	8 (8)	8 (8)
NistXTS749	4 (4)	4 (4)	8 (8)	14 (14)	8 (6)	14 (9)	12 (11)	18 (14)	14 (13)	20 (15)
Person	2 (2)	2 (2)	12 (12)	14 (14)	12 (7)	14 (8)	14 (9)	16 (10)	16 (11)	18 (12)
Products	6 (6)	6 (6)	26 (26)	28 (28)	26 (16)	28 (17)	35 (25)	37 (29)	37 (28)	39 (31)
RiskIt	24 (24)	24 (24)	48 (48)	72 (72)	48 (36)	72 (48)	69 (59)	93 (71)	71 (61)	95 (72)
StackOverflow	8 (8)	8 (8)	10 (10)	10 (10)	10 (9)	10 (9)	10 (9)	10 (9)	10 (9)	10 (9)
StudentResidence	4 (4)	4 (4)	14 (14)	16 (16)	14 (9)	16 (10)	20 (16)	22 (18)	20 (16)	22 (18)
UnixUsage	16 (16)	16 (16)	32 (32)	48 (48)	32 (24)	48 (32)	46 (42)	62 (50)	48 (44)	64 (51)
Usda	20 (20)	20 (20)	62 (62)	62 (62)	62 (41)	62 (41)	62 (41)	62 (41)	62 (41)	62 (41)
Total	316 (316)	316 (316)	958 (958)	1108 (1108)	958 (637)	1108 (712)	1177 (893)	1327 (1000)	1571 (1288)	1721 (1378)

(b) Column coverage criteria

Schema	UCC		AUCC		NCC		ANCC	
ArtistSimilarity	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)
ArtistTerm	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)
BankAccount	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (13)	18 (15)
BookTown	134 (134)	134 (134)	134 (134)	134 (134)	134 (134)	134 (134)	134 (132)	134 (132)
BrowserCookies	26 (26)	26 (26)	26 (26)	26 (26)	26 (26)	26 (26)	26 (24)	26 (24)
Cloc	20 (20)	20 (20)	20 (20)	20 (20)	20 (20)	20 (20)	20 (20)	20 (20)
CoffeeOrders	40 (40)	40 (40)	40 (40)	40 (40)	40 (40)	40 (40)	40 (30)	40 (33)
CustomerOrder	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (44)	64 (44)
DellStore	104 (104)	104 (104)	104 (104)	104 (104)	104 (104)	104 (104)	104 (73)	104 (73)
Employee	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)
Examination	42 (42)	42 (42)	42 (42)	42 (42)	42 (42)	42 (42)	42 (42)	42 (42)
Flights	26 (26)	26 (26)	26 (26)	26 (26)	26 (26)	26 (26)	26 (22)	26 (22)
FrenchTowns	28 (28)	28 (28)	28 (28)	28 (28)	28 (28)	28 (28)	28 (18)	28 (18)
Inventory	8 (8)	8 (8)	8 (8)	8 (8)	8 (8)	8 (8)	8 (8)	8 (8)
Iso3166	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (4)	6 (5)
iTrust	618 (618)	618 (618)	618 (618)	618 (618)	618 (618)	618 (618)	618 (544)	618 (564)
JWhoisServer	98 (98)	98 (98)	98 (98)	98 (98)	98 (98)	98 (98)	98 (60)	98 (60)
MozillaExtensions	102 (102)	102 (102)	102 (102)	102 (102)	102 (102)	102 (102)	102 (102)	102 (102)
MozillaPermissions	16 (16)	16 (16)	16 (16)	16 (16)	16 (16)	16 (16)	16 (16)	16 (16)
NistDML181	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (13)	14 (14)
NistDML182	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (50)	64 (64)
NistDML183	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)
NistWeather	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (13)	18 (15)
NistXTS748	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)
NistXTS749	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (14)	14 (13)	14 (13)
Person	10 (10)	10 (10)	10 (10)	10 (10)	10 (10)	10 (10)	10 (6)	10 (6)
Products	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (18)	18 (13)	18 (15)
RiskIt	114 (114)	114 (114)	114 (114)	114 (114)	114 (114)	114 (114)	114 (111)	114 (111)
StackOverflow	86 (86)	86 (86)	86 (86)	86 (86)	86 (86)	86 (86)	86 (85)	86 (85)
StudentResidence	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)	12 (12)	12 (11)	12 (11)
UnixUsage	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (64)	64 (62)	64 (62)
Usda	134 (134)	134 (134)	134 (134)	134 (134)	134 (134)	134 (134)	134 (113)	134 (113)
Total	1950 (1950)	1950 (1950)	1950 (1950)	1950 (1950)	1950 (1950)	1950 (1950)	1950 (1694)	1950 (1739)

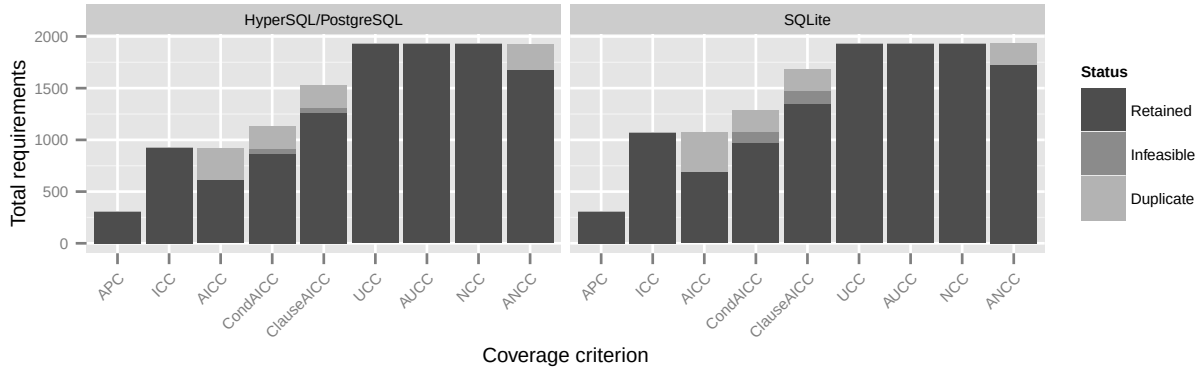


Figure 5.1: The number of infeasible, duplicate and retained test cases required by each coverage criterion for HyperSQL/PostgreSQL and SQLite.

The number of test cases generated for each schema, as well as the number remaining after removing those which were trivially infeasible and duplicate, are shown in Table 5.2. The bar plot in Figure 5.1 shows what proportion of those test cases removed are either infeasible or duplicate, relative to the number retained (i.e., not removed). As described previously, only simple cases of infeasibility such as $x = \texttt{NULL} \ \wedge \ x \neq \texttt{NULL}$ are currently detected automatically in SchemaAnalyst, therefore identifying and detecting more subtle kinds of infeasibility may increase the proportion of cases removed. Nonetheless, detecting this simple case prevents wasted computational effort by avoiding attempts to satisfy this type of infeasible combination of requirements.

The results show that for the constraint coverage criteria (APC, ICC, AICC, CondAICC and ClauseAICC), the number of test cases increases as the complexity of the criterion increases. As discussed in Section 3.5.1, each of the constraint coverage criteria builds on the definition of the previous criterion, adding further requirements, therefore explaining this trend. In the case of "Cloc", no tests were produced by any constraint coverage criterion because, as shown in Table 5.1, it includes no constraints and therefore contains nothing to test according to the definition of these criteria. Across all schemas, the two simplest criteria, APC and ICC, produced no test cases that were removed as trivially infeasible or duplicate, for either DBMS. Of the remaining constraint criteria, AICC has the greatest proportion of tests removed, followed by CondAICC then ClauseAICC – with 34%, 24% and 18% for PostgreSQL/HyperSQL, and 36%, 25% and 20% for SQLite.

For the column coverage criteria (UCC, AUCC, NCC, ANCC), the number of tests is

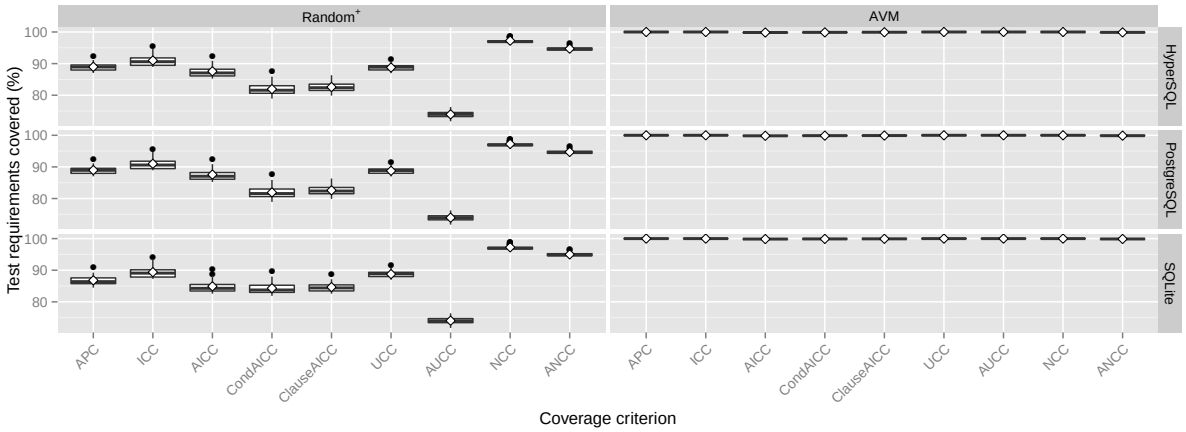


Figure 5.2: The percentage of test cases successfully generated by the AVM and Random$^+$ search algorithms, across all schemas for each DBMS. The boxes span between the 1st and 3rd quartile, with the line indicating the median and the white diamond showing the mean. The whiskers extend to the furthest data point up to 1.5× the inter-quartile range, with outliers beyond this marked as filled circles.

twice the number of columns for each schema as expected – for every schema, one case per column to requiring it to be `NULL` or `UNIQUE` (depending on the criterion), and one requiring it is not. Notably, of the four column coverage criteria only ANCC produces any test cases removed as trivially infeasible or duplicated. This is because ANCC may dictate a column must not be `NULL` when it already has an existing `NOT NULL` constraint defined for it. In the case of multiple `NOT NULL` constraints this produces tests that are duplicates of each other, such that only one should be retained.

Figure 5.2 shows how frequently the search algorithms were able to successfully generate data satisfying the predicates for the test cases, expressed as a percentage across all schemas and repeat trials for each DBMS. These results show that for all coverage criteria, the AVM algorithm is consistently more successful than Random$^+$, especially for the more complex constraint coverage criteria (i.e., CondAICC and ClauseAICC) and the active unique column coverage criterion (AUCC). Examining the results more closely, the few cases where the AVM algorithm was unable to generate data were caused by infeasible sets of predicates, which are not currently removed as with the trivially infeasible cases described previously. Automatically detecting such infeasible sets of predicates may be scope for future work. The Random$^+$ algorithm fails to successfully generate data for all test cases for any of the coverage criteria, with a general trend of failing more fre-

quently the more complex coverage criteria. However, contrary to this trend, Random$^+$ is more effective at generating data for the ClauseAICC criterion than the CondAICC criterion. This suggests the additional cases added by ClauseAICC are easier for the algorithm to generate test data for, leading to an overall increase in the success percentage. Despite this difference, statistical testing of the results for each schema using the Wilcoxon Rank-Sum test shows that the difference is not significant ($p < 0.05$) for either PostgreSQL/HyperSQL or SQLite, with values of 0.09 and 0.2 respectively.

Comparing the data generation success across DBMSs, Figure 5.2 shows that the Random$^+$ algorithm performs slightly better for PostgreSQL and HyperSQL than SQLite, although the difference is generally very small. This may be caused by PostgreSQL and HyperSQL disallowing `null` values for columns with a `PRIMARY KEY` constraint, meaning that generating data that violates a `PRIMARY KEY` constraint for these DBMSs is much easier for SQLite where the same value must be produced twice at random.

RQ1 Summary: The number of test cases produced for a given constraint coverage criteria increases as the complexity of the criterion increases, with the most complex (ClauseAICC) requiring over 4 times as many as the least complex (APC or ICC) when summing across all schemas. Due to differences in their model of SQL constraints, there is a slight increase in number of tests when using SQLite, compared to PostgreSQL and HyperSQL. When generating data for those test cases, the AVM algorithm is consistently more successful than the Random$^+$ algorithm, and is therefore the more effective choice. This result is unchanged by the choice of DBMS, where only Random$^+$ is affected, with slightly lower success when using SQLite.

RQ2: *What is the fault-finding capability of the test cases produced by each of the coverage criteria, and how does this depend upon the DBMS and search algorithm used?*

As described in Section 5.2.3, mutation analysis was used to assess the fault-finding capability of the test cases produced, as a measure of effectiveness of both the coverage criteria and data generation algorithms. The number of mutants for each schema, produced by applying the 14 mutation operators described in Chapter 4, is shown in Table 5.3. The values in brackets represent the actual number of mutants used for the mutation analysis process, after a number of unwanted mutants have been removed[3] ac-

[3]To briefly summarise, this removal process improves the accuracy of the results of mutation analysis, as well as reducing the execution time required, by analysing the intermediate representation of each mutant and checking against a series of patterns that identify these unwanted cases.

cording to automated techniques described next in Chapter 6. The differences between the number of mutants for PostgreSQL/HyperSQL and SQLite are a side effect of this removal process, as a consequence of their differing models of SQL constraints. After this process, a total of 3,775 mutants were generated for PostgreSQL and HyperSQL, and 3,915 mutants for use with SQLite. The box plots in Figure 5.3 shows the percentage of these mutants that were killed by the test cases produced by each coverage criterion, across all schemas. As depicted by the relatively short length of the "boxes", the variability across the 30 repeat trials was low for all both Random$^+$ and AVM search techniques, with maximum inter-quartile range values of 0.17% and 0.2%, respectively. This demonstrates that both algorithms generate similarly effective data and therefore indicates that the coverage criteria place suitably restrictive constraints on what data is required to satisfy their requirements. Figure 5.4 shows the mean percentage of mutants killed across each schema against the mean number of test cases produced, for every combination of coverage criterion, search algorithm and DBMS.

The results show that as the complexity of a constraint coverage criteria increases, so does the mutation score. Although this may be partly caused by the higher number of test cases produced for these criteria, therefore increasing the likelihood of killing mutants, Figure 5.4 shows that a greater number of test cases does not necessarily lead to a higher proportion of mutants killed. For example, ICC consistently produces more test cases than AICC and CondAICC, across all conditions, yet also kills fewer mutants than either in every case. Examining the results for both the unique column criteria and null column criteria shows similar results – for example, ANCC requires fewer test cases than NCC, but kills a higher proportion of mutants.

While the constraint coverage criteria are designed to specifically test the existing contents of the schema, in some cases the unique and null column coverage criteria killed a higher proportion of mutants. One explanation for this is that the mutation operators for `UNIQUE` and `NOT NULL` constraints (described in Chapter 4) produce a large proportion of mutants, as shown in Table 5.3, which the unique and null column coverage criteria are able to kill easily. Aggregating across all DBMSs and schemas these mutants represent 52% of total mutants, thus explaining the high proportion of mutants killed by the unique and null column coverage criteria.

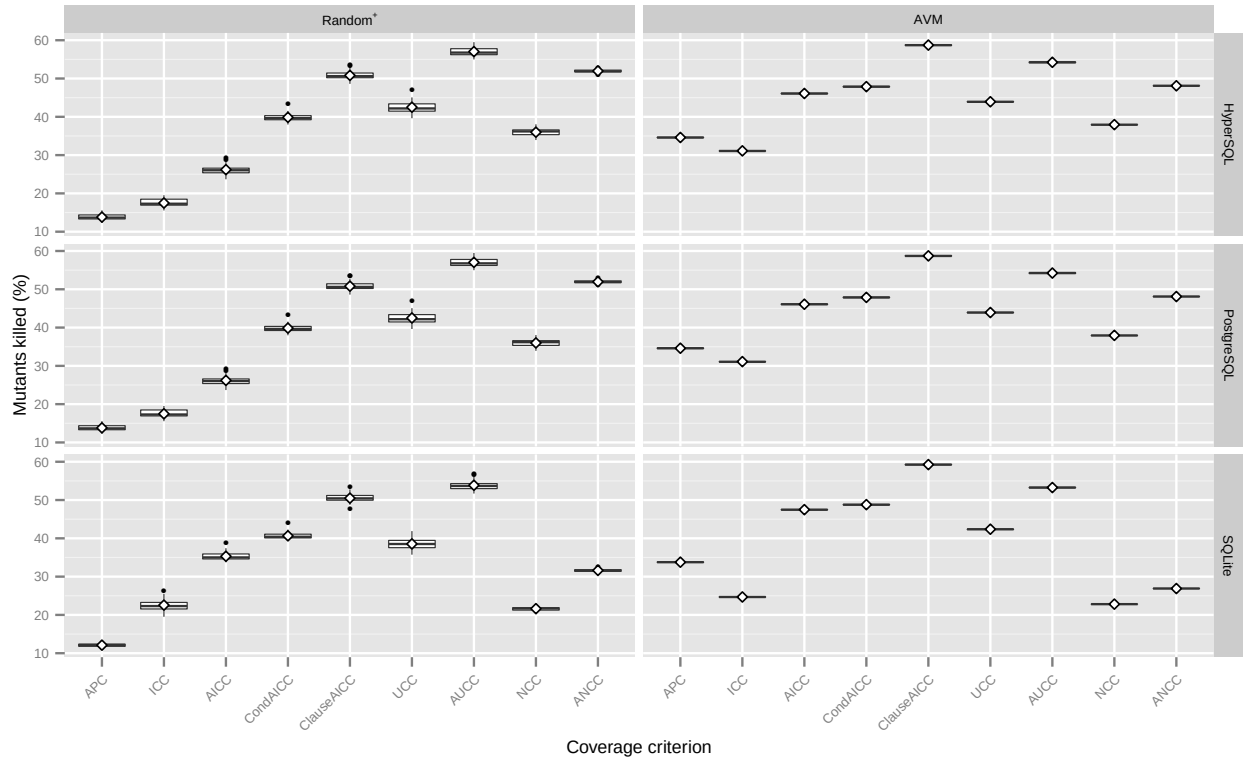


Figure 5.3: The proportion of mutants killed across all schemas, as a percentage of total number of mutants, for each combination of coverage criterion, DBMS and search algorithm used to generate data. The boxes span between the 1st and 3rd quartile, with the line indicating the median and the white diamond showing the mean. The whiskers extend to the furthest data point up to 1.5× the inter-quartile range, with outliers beyond this marked as filled circles.

Table 5.3: Numbers of mutants produced by the mutation operators for each schema.

The left and right columns are for PostgreSQL/HyperSQL and SQLite, respectively. The bracketed figures are the number of mutants remaining after a number of different types of unwanted mutants are automatically removed (see Chapter 6).

Schema	CInListElementR		CR		CRelOpE		FKCColumnPairA		FKCColumnPairE		FKCColumnPairR		NNCA		NNCR		PKCColumnA		PKCColumnE		PKCColumnR		UCColumnA		UCColumnE		UCColumnR		Total	
ArtistSimilarity	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (2)	2 (2)	2 (0)	2 (0)	3 (2)	3 (3)	0 (0)	0 (0)	2 (2)	2 (2)	0 (0)	0 (0)	1 (0)	1 (0)	3 (2)	3 (2)	0 (0)	0 (0)	0 (0)	0 (0)	13 (8)	13 (9)
ArtistTerm	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (4)	4 (4)	4 (4)	4 (4)	7 (4)	7 (7)	0 (0)	0 (0)	4 (4)	4 (4)	0 (0)	0 (0)	3 (0)	3 (0)	7 (4)	7 (4)	0 (0)	0 (0)	0 (0)	0 (0)	29 (20)	29 (23)
BankAccount	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6 (0)	6 (0)	10 (2)	10 (2)	1 (1)	1 (1)	4 (2)	4 (4)	5 (5)	5 (5)	7 (4)	7 (4)	7 (4)	7 (4)	2 (1)	2 (1)	9 (7)	9 (7)	0 (0)	0 (0)	0 (0)	0 (0)	51 (26)	51 (28)
BookTown	0 (0)	0 (0)	2 (2)	2 (2)	5 (5)	5 (5)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	52 (52)	52 (52)	15 (4)	15 (15)	56 (56)	56 (56)	27 (27)	27 (27)	11 (11)	11 (11)	67 (56)	67 (56)	0 (0)	0 (0)	0 (0)	0 (0)	235 (213)	235 (224)
BrowserCookies	0 (0)	0 (0)	2 (2)	2 (2)	15 (15)	15 (15)	8 (0)	8 (0)	36 (4)	36 (4)	2 (0)	2 (0)	9 (9)	9 (9)	4 (1)	4 (4)	10 (7)	10 (7)	13 (7)	13 (7)	3 (1)	3 (1)	18 (17)	18 (17)	15 (15)	15 (15)	3 (3)	3 (3)	138 (81)	138 (84)
Cloc	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	10 (10)	10 (10)	0 (0)	0 (0)	10 (10)	10 (10)	0 (0)	0 (0)	0 (0)	0 (0)	10 (10)	10 (10)	0 (0)	0 (0)	0 (0)	0 (0)	30 (30)	30 (30)
CoffeeOrders	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	11 (0)	11 (0)	25 (10)	25 (10)	4 (4)	4 (4)	10 (5)	10 (10)	10 (10)	10 (10)	15 (3)	15 (3)	15 (3)	15 (3)	5 (1)	5 (1)	20 (15)	20 (15)	0 (0)	0 (0)	0 (0)	0 (0)	115 (51)	115 (56)
CustomerOrder	2 (2)	2 (2)	1 (1)	1 (1)	0 (0)	0 (0)	10 (0)	10 (0)	18 (5)	18 (5)	7 (7)	7 (7)	5 (5)	5 (5)	27 (20)	27 (27)	25 (4)	25 (4)	25 (4)	25 (4)	7 (1)	7 (1)	32 (25)	32 (25)	0 (0)	0 (0)	0 (0)	0 (0)	159 (74)	159 (81)
DellStore	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	13 (13)	13 (13)	39 (39)	39 (39)	52 (52)	52 (52)	0 (0)	0 (0)	0 (0)	0 (0)	52 (52)	52 (52)	0 (0)	0 (0)	0 (0)	0 (0)	156 (156)	156 (156)
Employee	0 (0)	0 (0)	3 (3)	3 (3)	15 (15)	15 (15)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7 (6)	7 (7)	0 (0)	0 (0)	6 (6)	6 (6)	6 (6)	6 (6)	1 (1)	1 (1)	7 (6)	7 (6)	0 (0)	0 (0)	0 (0)	0 (0)	45 (43)	45 (44)
Examination	0 (0)	0 (0)	6 (6)	6 (6)	30 (30)	30 (30)	20 (0)	20 (0)	28 (6)	28 (6)	1 (1)	1 (1)	21 (19)	21 (21)	0 (0)	0 (0)	19 (14)	19 (14)	19 (14)	19 (14)	2 (1)	2 (1)	21 (19)	21 (19)	0 (0)	0 (0)	0 (0)	0 (0)	167 (110)	167 (112)
Flights	4 (4)	4 (4)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	6 (3)	6 (3)	2 (0)	2 (0)	7 (7)	7 (7)	6 (2)	6 (6)	9 (4)	9 (4)	18 (8)	18 (8)	4 (2)	4 (2)	13 (13)	13 (13)	0 (0)	0 (0)	0 (0)	0 (0)	70 (44)	70 (48)
FrenchTowns	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7 (0)	7 (0)	9 (1)	9 (1)	2 (2)	2 (2)	1 (1)	1 (1)	13 (13)	13 (13)	14 (14)	14 (14)	0 (0)	0 (0)	0 (0)	0 (0)	38 (31)	38 (31)	35 (20)	35 (20)	10 (3)	10 (3)	129 (85)	129 (85)
iTrust	273 (273)	273 (273)	8 (8)	8 (8)	0 (0)	0 (0)	2 (0)	2 (0)	5 (1)	5 (1)	1 (1)	1 (1)	221 (194)	221 (221)	88 (71)	88 (88)	265 (259)	265 (259)	240 (234)	240 (234)	44 (43)	44 (43)	309 (279)	309 (279)	0 (0)	0 (0)	0 (0)	0 (0)	1456 (1363)	1456 (1407)
Inventory	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (3)	4 (4)	0 (0)	0 (0)	3 (3)	3 (3)	3 (3)	3 (3)	1 (1)	1 (1)	6 (5)	6 (5)	3 (3)	3 (3)	1 (0)	1 (0)	21 (18)	21 (19)
Iso3166	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (1)	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	1 (1)	1 (1)	3 (2)	3 (2)	0 (0)	0 (0)	0 (0)	0 (0)	11 (9)	11 (10)
JWhoisServer	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (5)	5 (5)	44 (38)	44 (44)	43 (43)	43 (43)	43 (43)	43 (43)	6 (6)	6 (6)	49 (43)	49 (43)	0 (0)	0 (0)	0 (0)	0 (0)	190 (178)	190 (184)
MozillaExtensions	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	51 (49)	51 (51)	0 (0)	0 (0)	49 (49)	49 (49)	36 (36)	36 (36)	2 (2)	2 (2)	116 (114)	116 (114)	100 (100)	100 (100)	10 (9)	10 (9)	364 (359)	364 (361)
MozillaPermissions	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	8 (7)	8 (8)	0 (0)	0 (0)	7 (7)	7 (7)	7 (7)	7 (7)	1 (1)	1 (1)	8 (7)	8 (7)	0 (0)	0 (0)	0 (0)	0 (0)	31 (29)	31 (30)
NistDML181	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (0)	4 (0)	2 (0)	2 (0)	7 (5)	7 (7)	0 (0)	0 (0)	5 (4)	5 (4)	2 (0)	2 (0)	2 (0)	2 (0)	7 (7)	7 (7)	0 (0)	0 (0)	0 (0)	0 (0)	29 (16)	29 (18)
NistDML182	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3150 (0)	3150 (0)	15 (0)	15 (0)	32 (17)	32 (32)	0 (0)	0 (0)	17 (17)	17 (17)	0 (0)	0 (0)	15 (0)	15 (0)	32 (32)	32 (32)	0 (0)	0 (0)	0 (0)	0 (0)	3261 (66)	3261 (81)
NistDML183	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	14 (2)	14 (2)	2 (0)	2 (0)	6 (6)	6 (6)	0 (0)	0 (0)	6 (6)	6 (6)	0 (0)	0 (0)	0 (0)	0 (0)	7 (6)	7 (6)	2 (0)	2 (0)	2 (0)	2 (0)	40 (20)	40 (20)
NistWeather	0 (0)	0 (0)	5 (5)	5 (5)	0 (0)	0 (0)	6 (0)	6 (0)	11 (3)	11 (3)	1 (1)	1 (1)	4 (2)	4 (4)	5 (5)	5 (5)	6 (2)	6 (2)	8 (4)	8 (4)	3 (2)	3 (2)	9 (8)	9 (8)	0 (0)	0 (0)	0 (0)	0 (0)	58 (32)	58 (34)
NistXTS748	0 (0)	0 (0)	1 (1)	1 (1)	5 (5)	5 (5)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (2)	2 (2)	1 (1)	1 (1)	3 (3)	3 (3)	0 (0)	0 (0)	0 (0)	0 (0)	4 (4)	4 (4)	2 (2)	2 (2)	1 (1)	1 (1)	19 (19)	19 (19)
NistXTS749	0 (0)	0 (0)	1 (1)	1 (1)	5 (5)	5 (5)	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	1 (1)	4 (4)	4 (4)	3 (0)	3 (3)	4 (1)	4 (1)	5 (2)	5 (2)	3 (2)	3 (2)	7 (6)	7 (6)	0 (0)	0 (0)	0 (0)	0 (0)	33 (22)	33 (25)
Person	3 (3)	3 (3)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (5)	5 (5)	4 (4)	4 (4)	4 (4)	4 (4)	1 (1)	1 (1)	5 (4)	5 (4)	0 (0)	0 (0)	0 (0)	0 (0)	23 (22)	23 (22)
Products	0 (0)	0 (0)	4 (4)	4 (4)	20 (20)	20 (20)	0 (0)	0 (0)	4 (4)	4 (4)	2 (2)	2 (2)	4 (1)	4 (4)	5 (5)	5 (5)	5 (1)	5 (1)	6 (2)	6 (2)	4 (2)	4 (2)	9 (7)	9 (7)	0 (0)	0 (0)	0 (0)	0 (0)	63 (48)	63 (51)
RiskIt	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	125 (0)	125 (0)	157 (22)	157 (22)	10 (10)	10 (10)	42 (42)	42 (42)	15 (3)	15 (15)	45 (28)	45 (28)	40 (23)	40 (23)	12 (9)	12 (9)	57 (47)	57 (47)	0 (0)	0 (0)	0 (0)	0 (0)	503 (184)	503 (196)
StackOverflow	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	38 (38)	38 (38)	5 (5)	5 (5)	43 (43)	43 (43)	0 (0)	0 (0)	0 (0)	0 (0)	43 (43)	43 (43)	0 (0)	0 (0)	0 (0)	0 (0)	129 (129)	129 (129)
StudentResidence	0 (0)	0 (0)	3 (3)	3 (3)	15 (15)	15 (15)	1 (0)	1 (0)	3 (2)	3 (2)	1 (1)	1 (1)	4 (3)	4 (4)	2 (2)	2 (2)	4 (3)	4 (3)	4 (3)	4 (3)	2 (1)	2 (1)	6 (4)	6 (4)	0 (0)	0 (0)	0 (0)	0 (0)	45 (37)	45 (38)
UnixUsage	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	39 (0)	39 (0)	58 (12)	58 (12)	7 (7)	7 (7)	22 (22)	22 (22)	10 (2)	10 (10)	24 (7)	24 (7)	20 (3)	20 (3)	8 (3)	8 (3)	32 (26)	32 (26)	0 (0)	0 (0)	0 (0)	0 (0)	220 (82)	220 (90)
Usda	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	36 (36)	36 (36)	31 (31)	31 (31)	67 (67)	67 (67)	0 (0)	0 (0)	0 (0)	0 (0)	67 (67)	67 (67)	0 (0)	0 (0)	0 (0)	0 (0)	201 (201)	201 (201)
Total	282 (282)	282 (282)	38 (38)	38 (38)	110 (110)	110 (110)	236 (0)	236 (0)	3544 (83)	3544 (83)	67 (42)	67 (42)	640 (571)	640 (640)	335 (264)	335 (335)	831 (729)	831 (729)	550 (439)	550 (439)	144 (93)	144 (93)	1073 (968)	1073 (968)	157 (140)	157 (140)	27 (16)	27 (16)	8034 (3775)	8034 (3915)

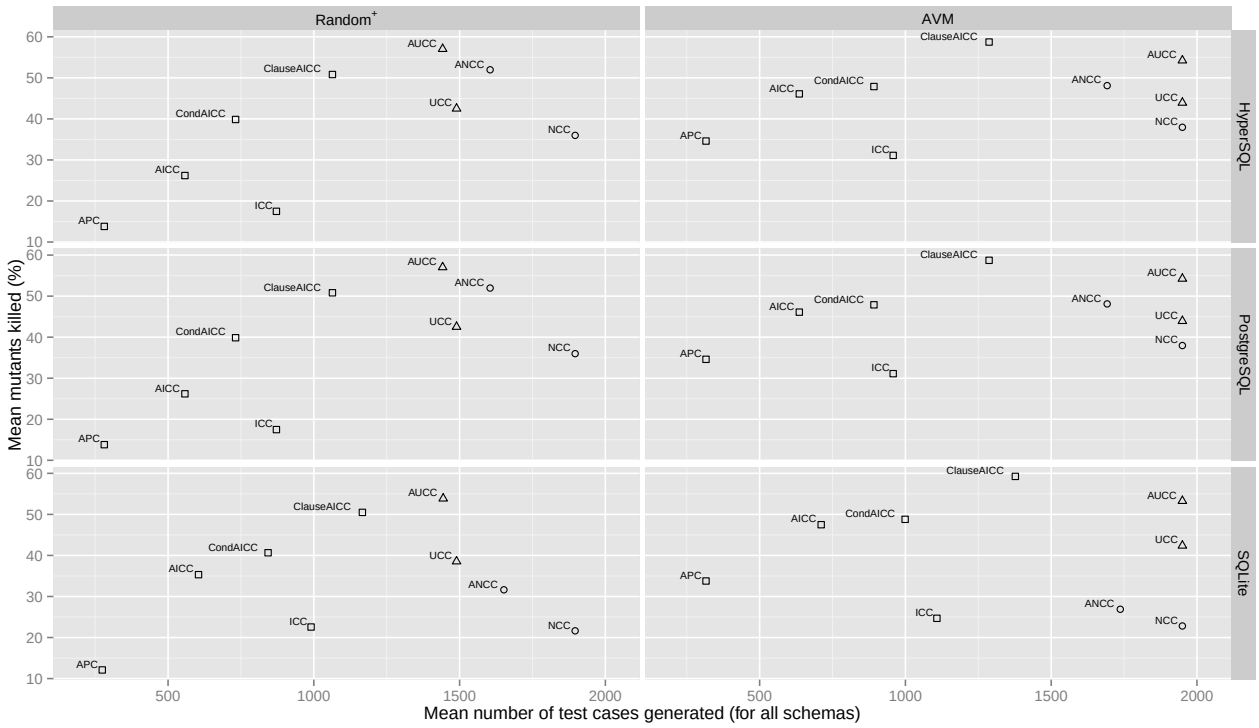


Figure 5.4: The mean percentage of mutants killed for each schema against the mean number of test cases generated. The points for constraint criteria, unique column criteria and null column criteria results are plotted as squares, triangles and circles, respectively.

Comparing the results between Random$^+$ and AVM, Figure 5.4 shows that the test suites generated for the constraint coverage criteria consistently kill fewer mutants. This is likely because the Random$^+$ algorithm does not successfully generate data for test cases as frequently, as shown in Figure 5.1, therefore reducing the likelihood that a given mutant will be killed. In contrast, when applying a unique or null column criterion the Random$^+$ appears to produce better data for test cases than the AVM technique. This result may be caused by the greater variety between test cases produced using a random search when compared to the data generated by the AVM technique, which is configured to start with the same set of 'initial' values for each test. By covering a larger area of the possible input domain, the test cases produced by Random$^+$ may be more likely to kill a wider range of mutants. Conversely, the test cases produced by the AVM technique are likely to be very similar to each other, as changes are only made from the initial values if they improve the progress towards satisfying the predicates for a given test case.

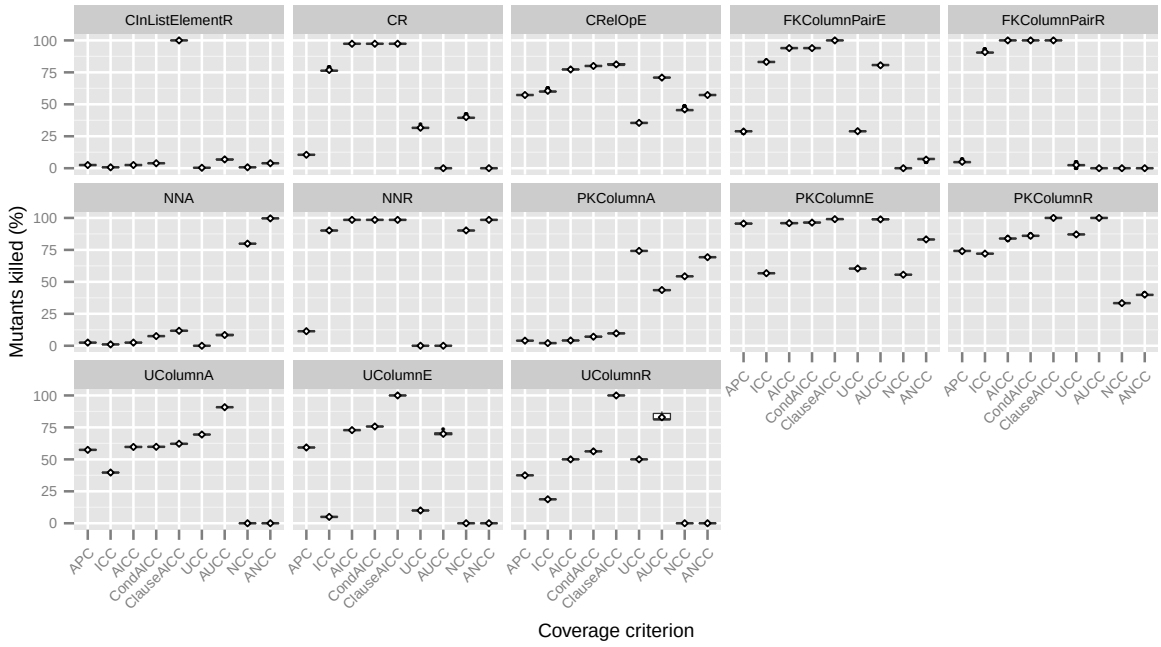
The criterion with the highest percentage of mutants killed depends upon which search algorithm is used to generate data. When using AVM, ClauseAICC produces test cases that kill the highest number of mutants, while for Random$^+$ AUCC is the most effective. However, overall the most mutants were killed when using ClauseAICC and the AVM algorithm, which resulted in a higher percentage of mutants killed than the AUCC criterion and the Random$^+$ algorithm. As the results for PostgreSQL and HyperSQL in Figure 5.3 do not clearly support this finding, the Wilcoxon Rank-Sum test was used to test for statistical significance, showing the difference is highly statistically significant ($p < 10^8$). For SQLite, the difference is clear because the boxes in Figure 5.3 do not overlap. Therefore, overall the test suites with the highest fault-finding capability are most likely to be those produced by the ClauseAICC criterion with data generated by the AVM search algorithm.

Comparing the results across DBMSs, there is not a difference between the results for PostgreSQL and HyperSQL, which is consistent with the findings of RQ1, where the same test cases were generated for both. However due to its differing model of SQL constraints, the results for SQLite differ slightly, especially for NCC and ANCC, which kill fewer mutants when using SQLite. This is because PostgreSQL and HyperSQL have an implicit `NOT NULL` constraint on the columns of a `PRIMARY KEY`, therefore NCC and ANCC (which set column values to `NULL`) are able to kill many mutants when using these DBMSs.

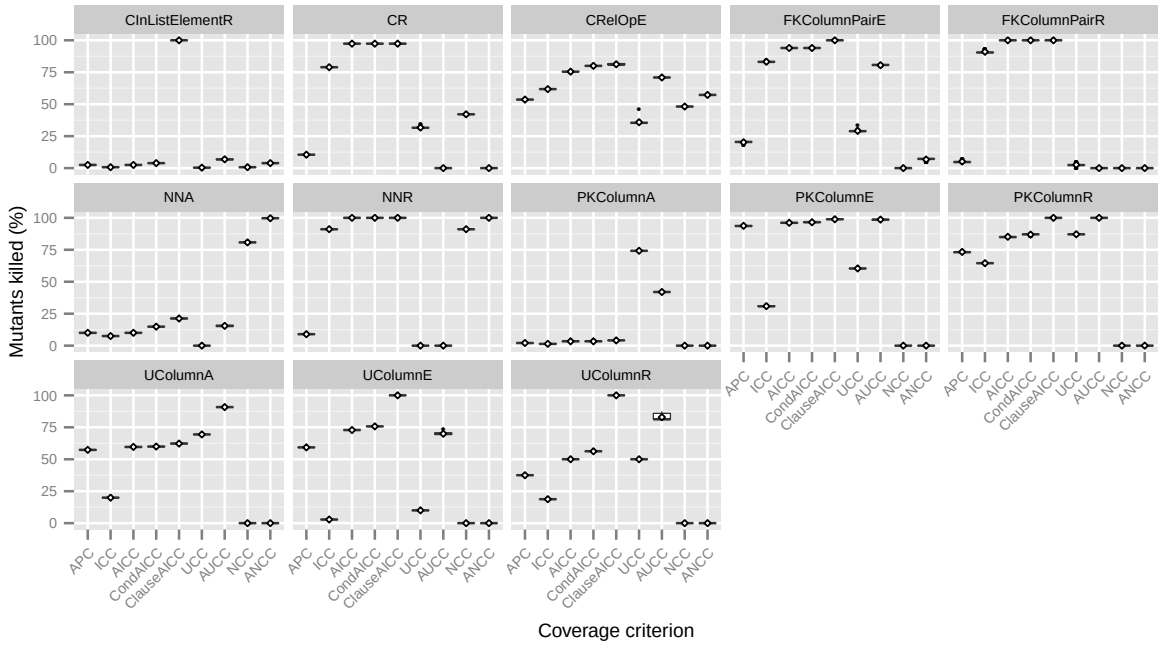
RQ2 Summary: The highest proportion of mutants are killed by the ClauseAICC criterion, if using the AVM search algorithm to generate data, or the AUCC criterion, if the Random$^+$ search algorithm is used. Overall, ClauseAICC and AVM produce the test cases that kill the most mutants. Although the DBMS choice affects the effectiveness of some coverage criteria, it does not alter this result.

RQ3: *Do the coverage criteria produce test cases that detect different types of faults, and how can these be characterised?*

The box plots of Figures 5.5 and 5.6 provide a breakdown of the percentage of mutants killed by each criterion according to the mutation operator used to produce them. As with the results discussed in RQ2, these plots show that the variability across trials is relatively small, as represented by the very small bars. The higher variability for Random$^+$ compared to AVM in this Figure is also consistent with the difference in success rate for producing test data shown in Figure 5.2.

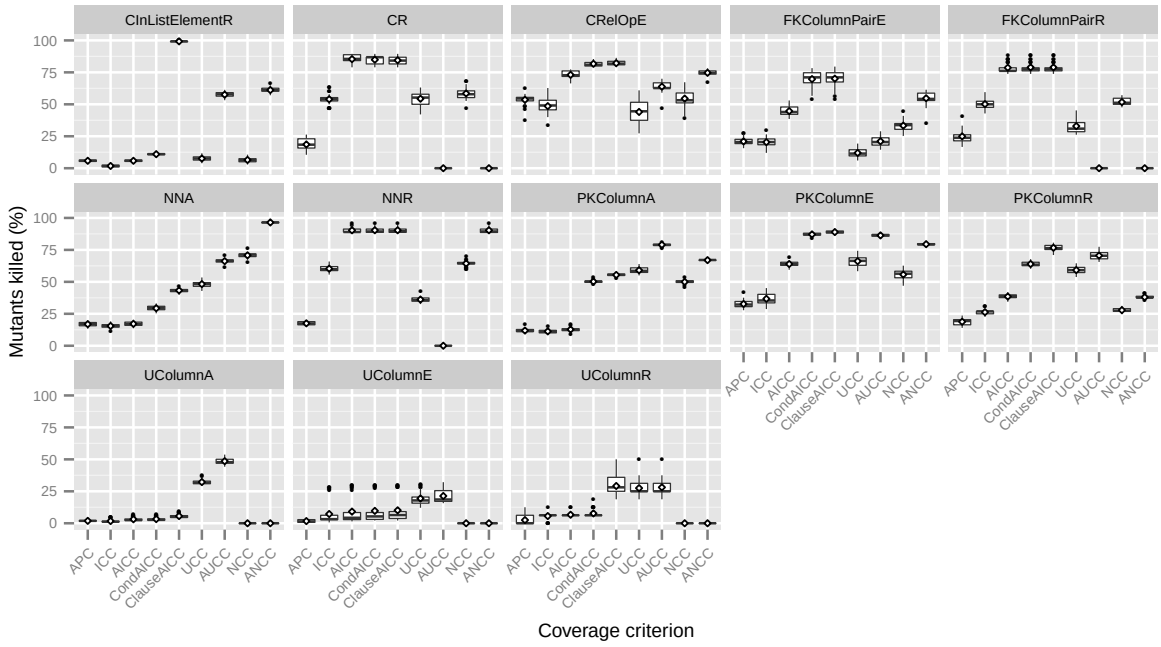


(a) Using the PostgreSQL/HyperSQL DBMSs

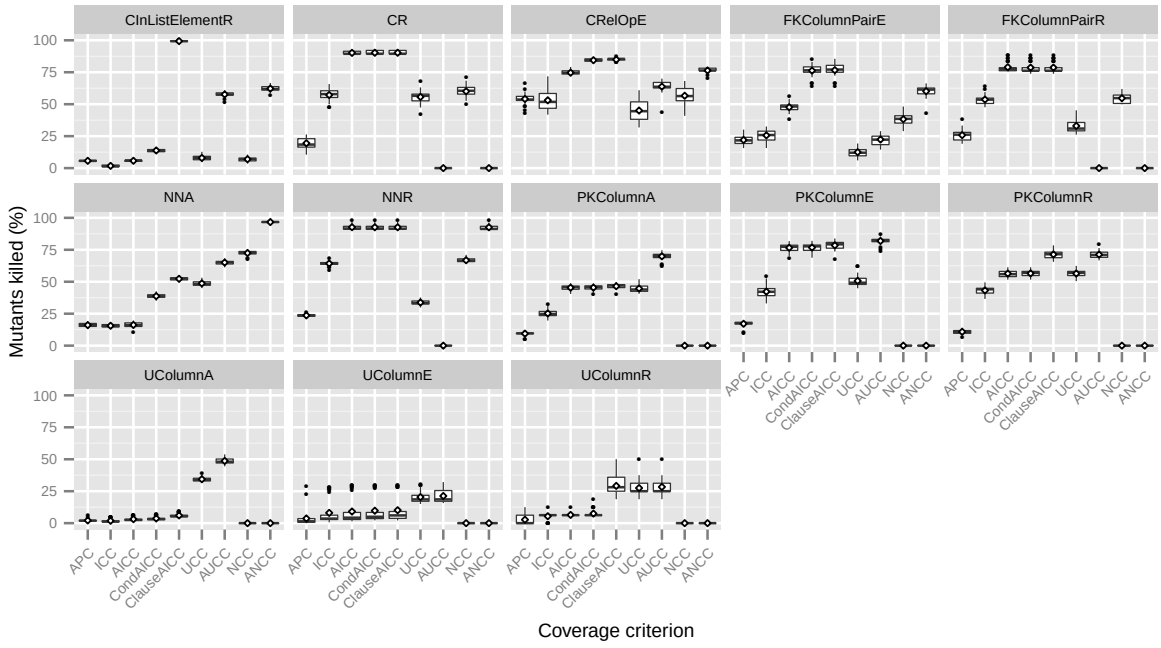


(b) Using the SQLite DBMS

Figure 5.5: The percentage of mutants killed using each coverage criterion and the AVM search algorithm, across all schemas, divided by the mutation operator used to produce them.



(a) Using the PostgreSQL/HyperSQL DBMSs



(b) Using the SQLite DBMS

Figure 5.6: The percentage of mutants killed using each coverage criterion and the Random$^+$ search algorithm, across all schemas, divided by the mutation operator used to produce them.

When using the null column coverage criteria, NCC and ANCC, a large proportion of mutants produced by those operators adding and removing `NOT NULL` constraints – NNA and NNR, respectively – were killed, as expected given the design of these criteria. In general, ANCC produces tests that kill the highest proportion of these mutants, equalling or bettering the results of other criteria. For mutants with added `NOT NULL` constraints the difference between ANCC and the next best criterion is particularly large. Similarly, the unique column coverage criteria, UCC and AUCC, kill many of the mutants produced by adding columns to existing `UNIQUE` constraints or creating new `UNIQUE` constraints – both created using the UColumnA operator. In particular, AUCC outperforms all other criteria in this respect, across all DBMSs and search algorithms. The high prevalence of mutants produced by the UColumnA operator, as shown in Table 5.3, combined with AUCC's high probability of killing such mutants partly explains why AUCC compares so favourably to other criteria in Figures 5.3 and 5.4, as discussed in RQ2.

The results for the constraint coverage criteria show that these are generally most likely to kill mutants produced by 'removal' and 'exchange' type operators (denoted with an 'R' or 'E' as the last character of the operator name, respectively). This trend applies generally across all types of constraint. For example, none of these criteria produce tests able to kill more than 10% of mutants produced by the PKColumnA operator, when using the AVM search algorithm and any DBMS, while nearly 100% of mutants from the PKColumnE and PKColumnR operators are killed in the best case. This is because the constraint coverage criteria test the constraints that are already present in the schema, and thus are able to differentiate the case of a mutant that removes or modifies one of these constraints.

From these results, the constraint coverage criteria can be described as being appropriate at testing for faults of "commission", where a mistake has been made in an existing constraint. In contrast, the null and unique column criteria are best suited to testing for faults of "omission", where a constraint has mistakenly not be included, as shown by the high degree of mutants they are able to kill that are produced by 'addition' operators, such as NNA and UColumnA.

There are some differences in results caused by the choice of DBMS, some of which may be explained by the differences in how PostgreSQL and HyperSQL handle `PRIMARY KEY` constraints, where the columns of the constraint are subject to an implicit `NOT NULL`

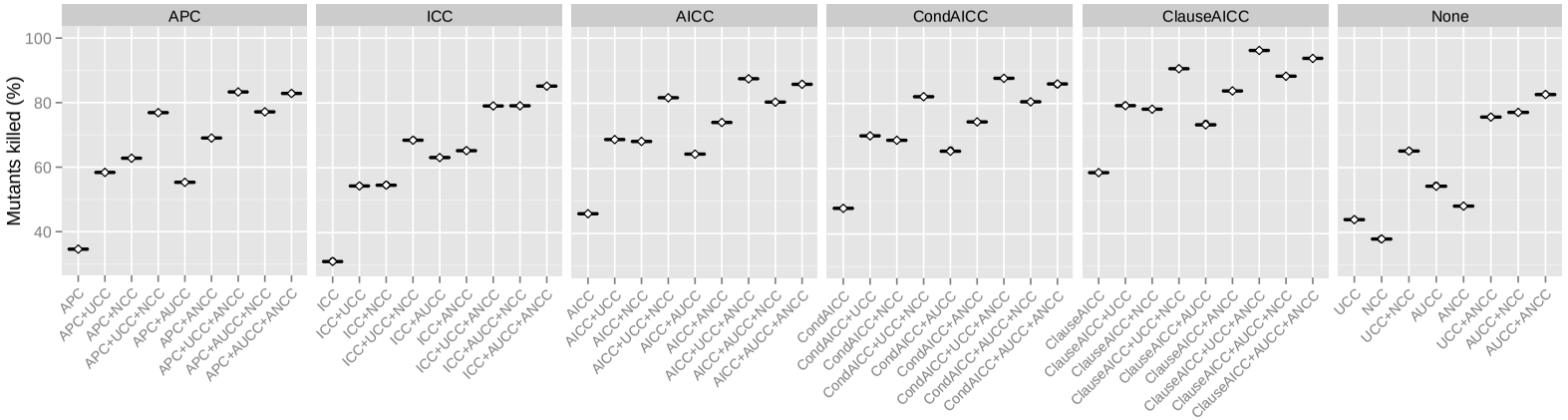
constraint, compared to SQLite, where there is no such restriction. This accounts for the higher proportion of `PRIMARY KEY` mutants – produced by PKColumnA, PKColumnE and PKColumnR – killed by the null column criteria, NCC and ANCC, when using PostgreSQL or HyperSQL rather than SQLite. In addition, the search algorithm used also affects the results. For example, the proportion of mutants killed by the constraint coverage criteria for the `UNIQUE` mutants – produced by UColumnA, UColumnE and UColumnR, is much better with data generated using the AVM algorithm than the Random$^+$ algorithm. This is likely because the AVM algorithm uses a default set of 'initial' values, whereas Random$^+$ chooses values randomly, thus the data values used are more likely to be the same between test cases. As a result, mutations of `UNIQUE` constraints are likely to be detected, and thus these mutants killed. Conversely, because Random$^+$ uses random values these are highly unlikely to be non-unique by chance, and therefore not likely to detect mutations of `UNIQUE` constraints.

RQ3 Summary: Constraint coverage criteria are generally better at detecting faults of commission where constraints are mistakenly included or affect a surplus of columns, as modelled by removal and exchange type mutation operators. The ClauseAICC criterion detects the highest proportion of such faults. The null and unique column criteria are generally the most successful at identifying faults of omission, where constraints have not been included, with the active criteria (i.e., ANCC and AUCC) finding the greatest number of these faults.

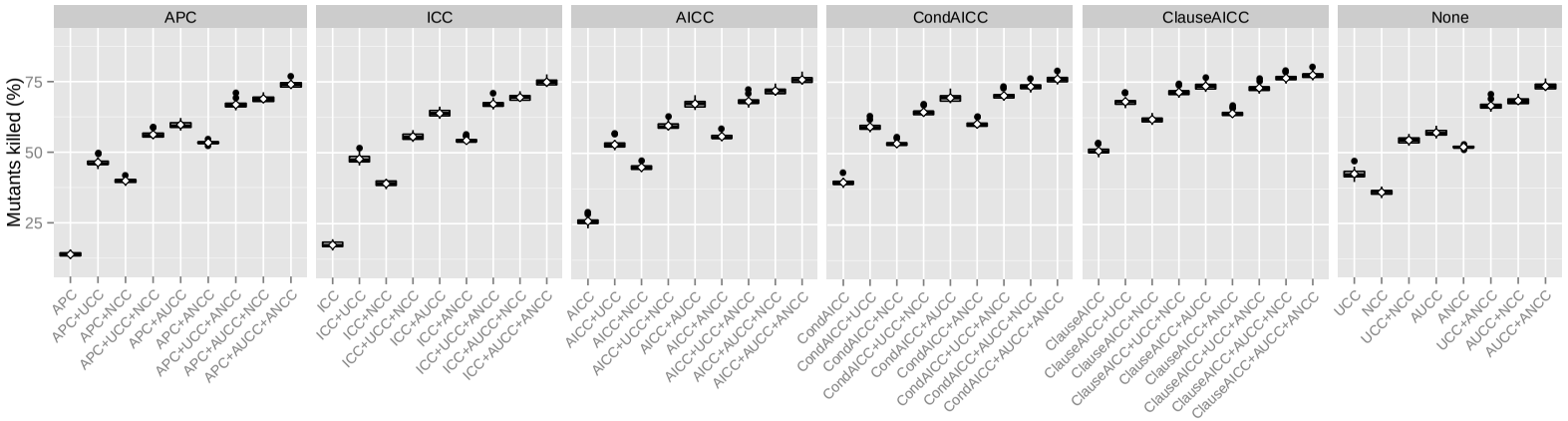
RQ4: *Can the tests from different types of coverage criteria be combined to improve the overall effectiveness?*

As discussed in RQ3, the various types of coverage criteria are able to detect different kind of faults with greater probabilities. Therefore this research question investigates whether combining the test cases produced by criteria from the different coverage types yields an increased fault-finding capability, as estimated by the proportion of mutants killed.

Figure 5.7 shows box plots of the percentage of mutants killed by each possible one, two and three-way combination of constraint coverage criterion, null column criterion and unique column criterion. This includes results for both the AVM and Random$^+$ search algorithms, as parts (a) and (b), and the PostgreSQL DBMS – results for other DBMSs were similar and are therefore omitted.



(a) Using the AVM search algorithm



(b) Using the Random$^+$ search algorithm

Figure 5.7: The percentage of mutants killed when combining different constraint and column coverage criteria, across all schemas, using the PostgreSQL DBMS (results for other DBMSs were similar and so are omitted).

When generating data with the AVM algorithm, the greatest proportion of mutants are killed by the combination of ClauseAICC + UCC + ANCC, with a mean average of 96%. This result is interesting as Figure 5.3 shows that when comparing AUCC and UCC alone, the former kills more mutants. By comparison, the combination of ClauseAICC + AUCC + ANCC yields a total of 94% of mutants killed, which is the second highest result. Using the Wilcoxon Rank-Sum test confirms that this difference is statistically significant ($p < 0.05$). However, the difference between combinations of UCC and AUCC is likely to be an artefact of how data is generated by the AVM algorithm, as ClauseAICC + AUCC + ANCC produces data killing the greatest proportion of mutants when instead using the Random$^+$ algorithm. As the AVM algorithm uses a default set of initial values, non-unique values will be used in test cases with a higher frequency than when using the Random$^+$ algorithm. For the UCC criterion, this may result in all columns being assigned non-unique values which may inadvertently cause the test cases generated to kill more mutants.

As the majority of mutants were killed by these combined sets of criteria, I was interested to examine which mutants remained, and therefore likely modelled more difficult to detect faults. After removing those mutants killed by test cases produced by the ClauseAICC + UCC + ANCC combination and AVM algorithm, all of the remaining live mutants were found to be produced by three operators – CRelOpE, PKColumnA and UColumnA. The CRelOpE (`CHECK` Relational Operator Exchange) operator produces mutants by swapping the operator (i.e., $=,\neq,<,\leq,\geq$) of a relational expression in a `CHECK` constraint with each other. Detecting these cases requires a large number of value to be produced (e.g., above, on and below a specified constant), especially if the variables on either side of the relational expression are column references. The PKColumnA and UColumnA operators add columns to existing `PRIMARY KEY` and `UNIQUE` constraints, respectively, or create new single column constraints of the corresponding types. The latter case is exhibited by many of the live mutants, suggesting there may be scope to further improve the column coverage criteria to detect such faults of omission. It is also worth noting that the survival of some mutants provides evidence that the faults modelled by the mutation operators test a schema more rigorously than the data generated by the coverage criteria is able to. Therefore to thoroughly test a schema it would be necessary to produce additional coverage criteria that exercise it more rigorously.

RQ4 Summary: Combining tests produced using constraint coverage, null column coverage and unique column coverage criteria does increase fault-finding capability, incorporating the detection of both faults of commission and omission. The highest proportion

of mutants were killed using combinations of ClauseAICC, ANCC and either UCC, if using the AVM algorithm, or AUCC, when using Random$^+$. In the best case, the mean percentage of mutants killed was 94%, a large improvement over the best single-criterion mean which was below 60%.

5.4 Summary

This Chapter described an experiment to evaluate the effectiveness of the different coverage criteria and search algorithms that form the data generation component of the SchemaAnalyst tool. The experiment included all nine coverage criteria, both available search algorithms, three DBMSs, and a range of 32 schemas, chosen for the wide range in their number of constraints. The results were then discussed in the context of four research questions. The first revealed that the more complex criteria, such as ClauseAICC, produce many more tests than the simple criteria, as well as that the coverage criteria result in a slightly higher number of tests being produced for SQLite, compared to PostgreSQL and HyperSQL. The results also showed that the AVM algorithm was consistently more successful in generating test cases than the Random$^+$ algorithm, producing test data in almost 100% of trials. In the second research question I investigated which coverage criterion had the highest fault-finding capability, measured by the number of mutants it managed to kill. The ClauseAICC criterion was found to be the most effective when the AVM algorithm was used to generate data, while the AUCC criterion killed slightly more mutants when using the Random$^+$ algorithm. Next, in RQ3 I analysed whether particular coverage criteria were better at killing mutants produced by specific mutation operators. This revealed that the constraint criteria were better at detecting faults of commission, while null and unique column criteria killed a higher proportion of mutants representing faults of omission. The ClauseAICC, ANCC and AUCC criteria were shown to have the highest fault finding capability. Finally, in RQ4 I investigated whether combining the test cases from each of the three types of criteria produced a test suite that detects a greater number of faults. This confirmed that such combined test suites can kill many more mutants than those produced by single criteria, with the best combination depending upon the data generation algorithm – ClauseAICC + ANCC + UCC when using the AVM algorithm, and ClauseAICC + ANCC + AUCC with the Random$^+$ algorithm. In the former case this killed 94% of mutants, which is much higher than the best single-criterion score of below 60%.

Chapter 6

Automatically Identifying Ineffective Mutants

6.1 Introduction

In Chapter 5, mutation analysis of relational database schemas was shown to be useful as a means of comparing different test suites, measuring their fault-finding capability according to the mutation operators of Chapter 4. However, as with mutation analysis of other software artefacts, not all mutants contribute meaningfully to the overall result obtained. This Chapter discusses three types of mutants – equivalent, redundant and quasi-mutants – classified as "ineffective", due to their detrimental impact on either the efficiency or accuracy of the results of mutation analysis. It is therefore beneficial to attempt to identify and remove these mutants prior to analysis.

First, each of these types of mutant are described in the context of relational database schemas, including example schemas and discussion of their effect on mutation analysis. Next, techniques and algorithms are discussed for automatically identifying these mutants, without the need for costly human involvement. Finally, these techniques are evaluated with an empirical study, to determine their efficacy and cost-effectiveness across a range of case study database schemas and multiple DBMSs.

In summary, the contributions this Chapter makes are:

1. Definitions and examples of equivalent, redundant and quasi-mutants, in the context of relational database schemas, and discussion of the negative impacts they have on the mutation analysis process;

2. Descriptions of a set of techniques which are able to identify each type of "ineffective" mutant in a fully automated way, including support for detecting equivalence between some different types of SQL constraint; and

3. An empirical evaluation of applying the techniques in practice to the mutation analysis of a variety of database schemas and three DBMSs, to determine how effective and efficient they are.

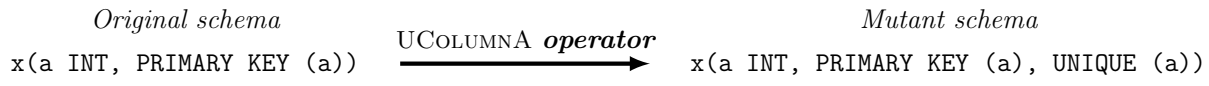
6.2 Types of Ineffective Mutant

This Section defines the three types of ineffective mutant, describes what circumstances cause them to be produced, provides an example, and the negative impact each has on mutation analysis of relational database schemas.

6.2.1 Equivalent mutants

While each of the database schema mutation operators, described in Chapter 4, makes a specific syntactic change to the original schema, it is nonetheless possible that a mutant may be produced that is functionally identical, but syntactically different, to the original. As a consequence, there exists no possible sequence of SQL `INSERT` statements that is accepted by the original schema and rejected by the mutant schema, or vice-versa. Such a mutant is commonly known as an *equivalent* mutant.

An example of an equivalent mutant is produced when a schema containing a `PRIMARY KEY` is mutated to contain a `UNIQUE` constraint over the same columns:



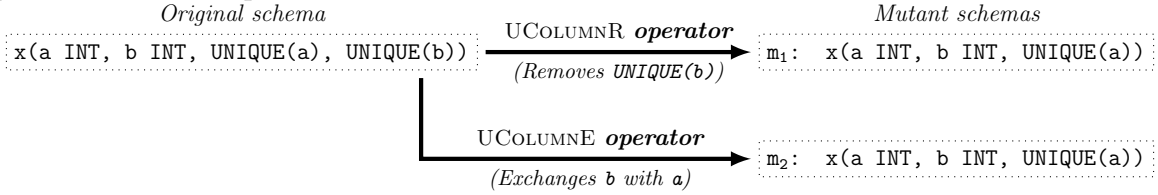
As both a `PRIMARY KEY` and `UNIQUE` constraint require the specified columns to contain unique values, the addition of `UNIQUE(a)` in the mutant schema does not alter the range of `INSERT` statements accepted, compared to the original schema. Therefore, the mutant is behaviourally identical to the original, thus is classified as an equivalent mutant.

Given that an equivalent mutant cannot be discerned from the original schema, during mutation analysis it will never be "killed". As a consequence, equivalent mutants may lead to inaccurate mutation scores, where test suites will be given lower-than-expected scores. This may in turn interfere with the comparison of different techniques for producing test suites, leading to false assessments of their effectiveness. In addition, in the practical application of mutation analysis each mutant left "alive" by a test suite is manually inspected, to determine in what way the test suite is deficient. As the number of equivalent mutants increases, this human cost can quickly become prohibitively expensive. In addition, for every equivalent mutant present, the full test suite will be executed against it, in an ineffectual attempt to kill it, thus needlessly adding to the computational cost of mutation analysis.

To mitigate these issues, it is important to identify equivalent mutants such that they can be removed prior to mutation analysis. As the number of mutants produced may grow very large for complex schemas, thus rendering human detection of equivalence infeasibly expensive, it is necessary for this process to be automated as much as is possible.

6.2.2 Redundant mutants

Where two database schema mutants are behaviourally the same as each other they are classified as *redundant* mutants. This implies that for any sequence of SQL `INSERT` statements the acceptance pattern will be the same for both mutants. Redundant mutants may be produced when Exchange-type mutation operators make a change that effectively removes a constraint, thus overlapping with the changes made by a Removal-type operator. For example:



In the case of the m_2, the mutation causes there to be two `UNIQUE(a)` constraints. However, as there is no semantic difference between this and the original single `UNIQUE` constraint, this is simplified such that the two mutants are identical, both behaviourally and syntactically.

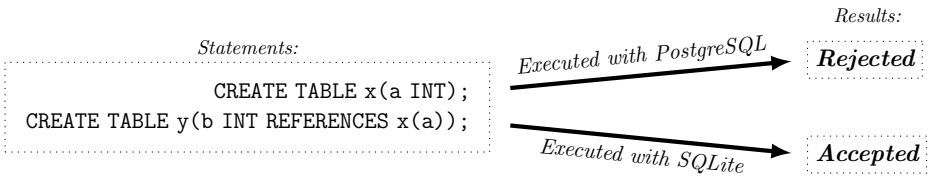
Redundant mutants may interfere with the comparison of different test suite generation techniques, by biasing the mutation scores obtained towards particular types of mutants, thus causing the score to potentially either over- or under-estimate the true effectiveness of the test suites. In addition, as sets of redundant mutants are behaviourally equivalent, with respect to each other, executing the test suite against each is an unnecessary computational cost.

As with equivalent mutants, it is therefore also important to detect the presence of redundant mutants, prior to mutation analysis, to ensure results are both accurate and obtained efficiently. Given each possible pair of mutants may be redundant with respect to each other, it is impractical to consider identifying redundant mutants manually, thus necessitating an automated approach.

6.2.3 Quasi-mutants

While there is an official ISO standard for the SQL language [7], defining how an SQL relational database should behave, each DBMS may vary in its interpretation and conformance to this standard [1]. As a consequence, it is possible to produce a mutant that is syntactically valid for one DBMS, yet invalid for another. A syntactically invalid mutant is usually described as being "still-born", in mutation analysis of programs. However, because these mutants are only still-born with respect to certain DBMSs they are referred to as *quasi-mutants* instead.

A quasi-mutant exposes a specific difference between the internal models of SQL used by at least two DBMSs. For example, in a PostgreSQL database for every `FOREIGN KEY` there must be a `PRIMARY KEY` or `UNIQUE` constraint defined for the reference columns, otherwise the `CREATE TABLE` statement will be rejected. However, submitting a similar statement to an SQLite database causes no such error. In this case, the mutants are classified as quasi-mutants with respect to the PostgreSQL DBMS. For example, the following schema would be rejected by PostgreSQL but accepted by SQLite:



Although quasi-mutants do not stop the application of mutation analysis, and can trivially be prevented from affecting the overall mutation score, identifying these mutants ahead of time may allow for the use of optimisations otherwise precluded by their presence [117]. In the interest of efficiency, it is therefore beneficial to detect and discard any quasi-mutants specific to the DBMS in use, prior to mutation analysis.

6.3 Detecting Ineffective Mutants Automatically

The content in this Section details techniques to detect each of the types of ineffective mutant described in Section 6.2, which would otherwise detrimentally affect mutation analysis of relational database schemas. These techniques work automatically, thus reducing the amount of costly human inspection required, decreasing the overall cost of mutation analysis.

All of the techniques in this Section operate directly on the intermediate representation of SQL used internally in the SchemaAnalyst tool, as described in Section 3.3, and which is both produced when parsing SQL schemas for use with the tool and used directly as the representation for mutants. This enables the schemas and their mutants to be analysed statically in order to identify ineffective mutants, prior to mutation analysis.

6.3.1 Equivalent mutants

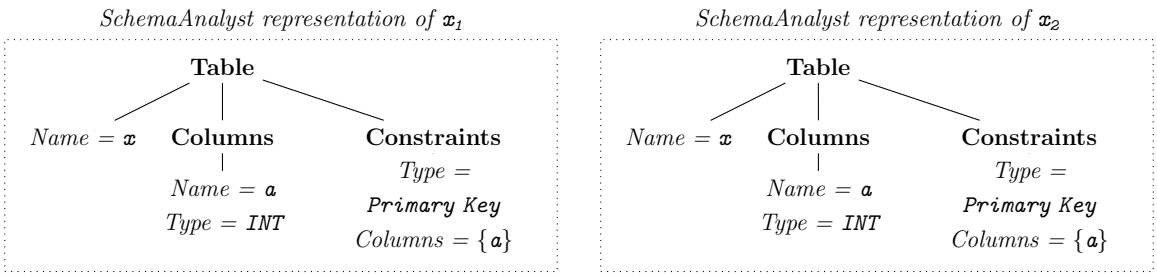
As defined previously in Section 6.2.1, a mutant is classified as equivalent if it is functionally identical, but syntactically different, to the original schema. All equivalent mutants may be categorised as being either *structurally* or *behaviourally* equivalent to the original schema. These two categories require different means of detection, which are described in turn in this Section.

Structural equivalence

Two schemas are *structurally equivalent* to one another if they are identical, ignoring small possible differences in the SQL used to define them – for example, whether a `PRIMARY KEY` is defined as part of a column definition, as shown by tables $\texttt{x}_1$ and $\texttt{x}_2$ respectively:

```
CREATE TABLE x1 (a INT PRIMARY KEY)
CREATE TABLE x2 (a INT, PRIMARY KEY(a))
```

Such small differences are factored away when parsing a schema into the SchemaAnalyst intermediate representation, which can be approximately visualised as the following trees of objects for this example:



By visual inspection of these trees it is easy to observe that they are identical, even though the original SQL definitions of each differ subtly, and therefore that the initial schemas are structurally equivalent.

To automate this comparison, thus eliminating the human cost, the functions of Figure 6.1 are applied to the original schema and each mutant schema, by first invoking the SCHEMAEQUIV function. Together, these functions compare each attribute of two schemas in turn, starting at the top of the trees of objects and stopping if any difference is detected between the two. This allows the detection of structurally equivalent mutants to be fully automated, so they can be removed prior to mutation analysis, reducing the cost and improving the accuracy of the resulting mutation score.

Behavioural equivalence

Although two schemas may not be structurally equivalent, it is still possible that they will behave identically for all `INSERT` statements and are therefore classified as *behaviourally*

```
function schemaEquiv(a, b)
      return (name(a) = name(b)
          and |tables(a)|=|tables(b)|
          and tablesEquiv(tables(a),tables(b))
          and primaryKeysEquiv(getPrimaryKey(a),getPrimaryKey(b))
          and foreignKeysEquiv(getForeignKeys(a),getForeignKeys(b))
          and uniquesEquiv(getUniques(a),getUniques(b))
          and checksEquiv(getChecks(a),getChecks(b))
          and notNullsEquiv(getNotNulls(a),getNotNulls(b)))

function tablesEquiv(a, b)
      return (name(a) = name(b)
          and columnsEquiv(getColumns(a),getColumns(b)))

function columnsEquiv(a, b)
      return (name(a) = name(b)
          and getType(a) = getType(b))

function primaryKeysEquiv(a, b)
   return multicolumnConstraintEquiv(a,b)

function foreignKeysEquiv(a, b)
      return (multicolumnConstraintEquiv(a,b)
          and name(getRefTable(a)) = name(getRefTable(b))
          and |getRefColumns(a)| = |getRefColumns(b)|
          and getRefColumns(a) ⊆ getRefColumns(b))

function uniquesEquiv(a, b)
    return multicolumnConstraintEquiv(a,b)

function checksEquiv(a, b)
      return (name(a) = name(b)
          and name(getTable(a)) = name(getTable(b))
          and expression(a) = expression(b))

function notNullsEquiv(a, b)
      return (name(a) = name(b)
          and name(getTable(a)) = name(getTable(b))
          and name(column(a)) = name(column(b)))

function multicolumnConstraintEquiv(a, b)
      return (name(a) = name(b)
          and name(getTable(a)) = name(getTable(b))
          and |getColumns(a)| ≠ |getColumns(b)|
          and getColumns(a) ⊆ getColumns(b))
```

Figure 6.1: The set of functions used to detect structural equivalence, using the utility functions described in Figure 4.1 (see page 108).

Algorithm 15 Detecting `NOT NULL` equivalent to `CHECK`

```
function notNullEquivToCheck(a, b)
    unmatched ← ∅
    for all notNullA in getNotNulls(a) do
        found ← F
        for all notNullB in getNotNulls(b) do
            if notNullA = notNullB then
                found ← T
                break
        if found = F then
            unmatched ←unmatched ∪ {notNullA}
    if |unmatched| = 0 then return T
    else
        for all notNull in unmatched do
            if getCheck((getColumns(notNull)) is not null) ∉ getChecks(b) then return F
    return T
```

equivalent. This is due to the overlapping effects various SQL features have on the acceptance of data into the database, meaning the same overall effect can be achieved in multiple ways by applying different constraints. This Section describes three "patterns" that describe such sets of overlapping constraints, which can be automatically identified using a corresponding set of detection functions, implemented in the SchemaAnalyst tool. Where the pattern is only applicable to a particular DBMS, due to differences in their interpretation and implementation of the SQL specification, this is detailed.

Pattern 1: `NOT NULL` in `CHECK` constraints. Defining a `NOT NULL` constraint for a column, `x`, is behaviourally equivalent to including a `CHECK` constraint with the expression `x IS NOT NULL`. For example, tables $\texttt{x}_1$ and $\texttt{x}_2$ will accept and reject exactly the same `INSERT` statements and are therefore behaviourally equivalent:

```
CREATE TABLE x1 (a INT NOT NULL)
CREATE TABLE x2 (a INT, CHECK(a IS NOT NULL))
```

To detect this behavioural equivalence, the notNullsEquiv function in Figure 6.1 must be modified to identify whether each `NOT NULL` constraint in schema `a` has a corresponding `NOT NULL` constraint or behaviourally equivalent `CHECK` constraint in schema `b`. This is implemented using the algorithm in Algorithm 15 as the function notNullEquivToCheck.

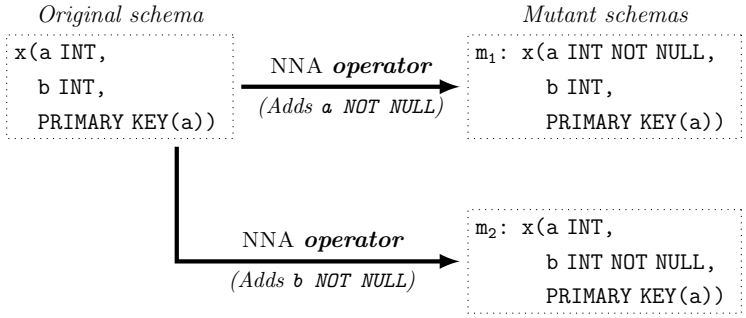
Pattern 2: `NOT NULL` and `PRIMARY KEY` constraints. If the DBMS in use is either PostgreSQL or HyperSQL, a `PRIMARY KEY` constraint specifies not only that the value(s)

Algorithm 16 Detecting `NOT NULL` and `PRIMARY KEY` overlap

```
function notNullOnPrimaryKey(schema)
    for all t do in getTables(schema)
        for all nn do in getNotNulls(t)
            if getPrimaryKey(getColumns(nn)) ∈ primaryKeys(t) then
                getNotNulls(t) ← createNotNulls(t) / {nn}
```

for the specified column(s) must be unique, but also cannot be `NULL`. In contrast, SQLite does not enforce this restriction, allowing `NULL` values to be used unless an additional `NOT NULL` constraint is defined for each of the columns.

As a consequence, the `NOT NULL` addition operator (NNA, Algorithm 10) may produce mutants that are behaviourally indistinguishable from the original schema, when using either PostgreSQL or HyperSQL, by adding a `NOT NULL` constraint to a column already in a `PRIMARY KEY`. For example, given the table `x` and the mutants m_1 and m_2, both produced by the NNA operator:



In this case, m_1 contains a `NOT NULL` constraint overlapping the `PRIMARY KEY`, as both are defined for the same column, `a`. As a result, there is no `INSERT` statement that will be accepted by the original schema and rejected by m_1, or vice versa. In contrast, m_2 does not have such an overlap, and therefore an `INSERT` can be used to identify the mutation – in this case, the values `(a=1,b=NULL)` would suffice, being accepted by m_2 but rejected by the original. Therefore, it can be said that m_1 is behaviourally equivalent, while m_2 is not.

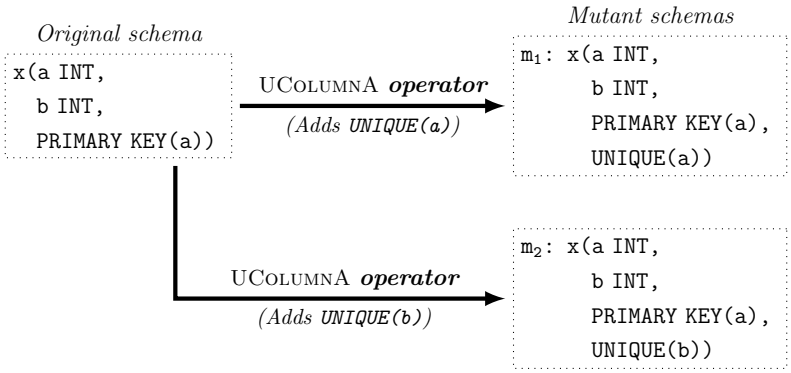
To detect these mutants, the notNullOnPrimaryKey function in Algorithm 16 is applied, prior to the application of the structural equivalence detection functions. This function analyses all tables in schema and removes any `NOT NULL` constraint that overlaps with the columns of the `PRIMARY KEY`, a process which can be seen as being analogous

Algorithm 17 Detecting `UNIQUE` and `PRIMARY KEY` overlap

```
function uniqueOnPrimaryKey(schema)
    for all t do in getTables(schema)
        for all uc do in getUniques(t)
            if getPrimaryKey(getColumns(uc)) ∈ primaryKeys(t) then
                getUniques(t) ← createUniques(t) / {uc}
```

to removing 'dead code' that does not affect behaviour from a program. This removal process renders mutants such as $\texttt{m}_1$ structurally equivalent to the original schema, enabling them to be identified and removed automatically.

Pattern 3: `UNIQUE` and `PRIMARY KEY` overlap. Where a `PRIMARY KEY` and `UNIQUE` constraint are defined over the same set of columns, the `UNIQUE` constraint is superfluous, as the `PRIMARY KEY` already enforces the uniqueness of those columns, such that the `UNIQUE` does not impose any additional restrictions on the data accepted by the database. Therefore, a mutant whose mutation is the addition of a `UNIQUE` constraint matching the existing `PRIMARY KEY` constraint is a behaviourally equivalent mutant. For example, given the table `x` and the mutants $\texttt{m}_1$ and $\texttt{m}_2$, both produced by the `UNIQUE` column addition (UColumnA) operator:



In this case, the `UNIQUE` constraint added in $\texttt{m}_1$ does not alter the acceptance behaviour compared to the original schema, because it contains the same set of columns as the `PRIMARY KEY`. This implies there is no possible sequence of `INSERT` statements that will be accepted by $\texttt{m}_1$ and rejected by the original, or vice versa. In contrast, the mutation in $\texttt{m}_2$ does not contain the same set of columns as the existing `PRIMARY KEY` and thus the mutation can be detected – for example, the sequence of values $\{\texttt{(}a\texttt{=1,}b\texttt{=1),(}a\texttt{=2,}b\texttt{=1)}\}$ will be accepted by the original schema but rejected by $\texttt{m}_2$.

As with pattern 2, the automatic detection of these behaviourally equivalent mutants applies a function to remove the unneeded constraint(s) – uniqueOnPrimaryKey in Algorithm 17 – and then utilizes the structural equivalent mutant detection functions to identify and discard the mutant. Applying this process to $\texttt{m}_1$, given above, would cause the `UNIQUE(a)` constraint to be removed, leaving the mutant structurally identical to the original schema.

6.3.2 Redundant mutants

As defined previously in Section 6.2.2, a mutant is a redundant if it is behaviourally identical to another mutant. These mutants may impact on the ability to use mutation analysis to compare different data generation techniques by biasing the mutation scores towards particular types of mutant, as well as increasing the time needed for mutation analysis, and should therefore be removed.

Identification of redundant mutants is achieved by applying the automatic equivalent mutant detection approaches described in Section 6.3.1, used there to compare each mutant against the original schema, to compare each pair of mutants instead. By including the techniques for identifying both structural and behavioural equivalence, this enables the detection of either type of redundant mutant.

6.3.3 Quasi-mutants

Due to slight differences between DBMSs, it is possible a mutant is syntactically valid for one DBMS yet invalid for another; Section 6.2.3 classifies these mutants as quasi-mutants. The presence of these mutants may affect the efficiency of mutation analysis, therefore it is beneficial to automatically identify and remove them. However, this process should only be applied when they are invalid with respect to the DBMS currently being used, otherwise the accuracy of the mutation score will be affected.

Applying the mutation operators discussed previously in Chapter 4 to mutation analysis using three DBMSs – PostgreSQL, HyperSQL and SQLite – has led to the discovery of one pattern that describes a quasi-mutant, which is invalid when either PostgreSQL or HyperSQL are used:

$$\forall fk \in \textsc{getForeignKeys}(t) \implies$$
$$\exists \textsc{getUnique}(\textsc{getRefColumns}(fk)) \in \textsc{getUniques}(t) \ \vee$$
$$\exists \textsc{getPrimaryKey}(\textsc{getRefColumns}(fk)) \in \textsc{getPrimaryKey}(\textsc{getRefTable}(fk))$$

That is, for every foreign key constraint, the columns referred to in the reference table must either be subject to a `PRIMARY KEY` or `UNIQUE` constraint, otherwise the schema is invalid and will be rejected when a `CREATE TABLE` statement is issued, preventing a database from being created. If a `CREATE` statement violating this pattern is submitted to PostgreSQL an error message is returned stating:

"There is no unique constraint matching given keys for referenced table (Error 42830 (invalid foreign key))"

...while HyperSQL produces a comparable error message:

"a UNIQUE constraint does not exist on referenced columns (Error 5529)"

Because a DBMS will reject a `CREATE` statement if it is syntactically invalid, the DBMS itself can be used to identify quasi-mutant. This simply involves submitting the schema for each mutant to the intended DBMS as `CREATE TABLE` statements and examining whether it was accepted or rejected. However, this may become very costly as the size of the schema and number of mutants increases, as `CREATE TABLE` statements are expensive operations. In addition, between each mutant a series of `DROP TABLE` statements must be executed, provided at least one table of the prior mutant was accepted and a database created. To ameliorate this cost, the submission of each mutant schema may be wrapped in an SQL transaction, such that the transaction can be "rolled back" between mutants to return the database to an empty state.

Alternatively, it is possible to detect quasi-mutants by statically analysing the schema, thus avoiding the need for DBMS interaction which may prove computationally costly. I achieved this by transforming the more formal description of the quasi-mutant pattern into the detection function, detectQuasi, given in Algorithm 18. By applying this function to each mutant schema prior to mutation analysis, provided either PostgreSQL or HyperSQL are the DBMS in use, any quasi-mutants matching this known pattern can be identified both automatically and without interaction with the DBMS. These can

Algorithm 18 Detecting quasi-mutants

```
function detectQuasi(schema)
    for all t in getTables(schema) do
        for all fk in getForeignKeys(t) do
            if getPrimaryKey(getRefColumns(fk)) ∉ primaryKeys(getRefTable(fk))
                and getUnique(getRefColumns(fk)) ∉ getUniques(getRefTable(fk)) then
                    return T
    return F
```

then be discarded, allowing the mutation analysis process to be implemented assuming all remaining mutants are syntactically valid for the chosen DBMS.

Although currently only one pattern of quasi-mutant is identified using this detection function, to date experimentation with the SchemaAnalyst tool has not led to the discovery of any additional patterns, using the three currently supported DBMSs, the mutation operators of Chapter 4 and over 50 schemas. However, I have designed the mutation analysis process implemented in the SchemaAnalyst tool to automatically report when the assumption that "all mutants remaining after quasi-mutant removal must be syntactically valid" is violated. Any mutant that violates this constraint must represent a new pattern of quasi-mutant, therefore if additional DBMSs or operators are added these mutants can be identified. These new patterns can then be encoded within detection functions and easily integrated into the mutation analysis system of SchemaAnalyst, applied selectively depending on the DBMS in use.

6.4 The Impact of Ineffective Mutant Removal

This Section describes the design and results of an empirical study aiming to determine what effect removal of ineffective mutants has, applying the techniques described in Section 6.3. This is measured both in terms of the change in the resulting mutation score and the time taken for mutation analysis, including the additional time required to detect and remove them.

Table 6.1: Schemas analysed in the empirical study

Schema	Tables	Columns	Checks	Foreign Keys	Not Nulls	Primary Keys	Uniques	∑Constraints
ArtistSimilarity	2	3	0	2	0	1	0	3
ArtistTerm	5	7	0	4	0	3	0	7
BankAccount	2	9	0	1	5	2	0	8
BookTown	22	67	2	0	15	11	0	28
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
Flights	2	13	1	1	6	2	0	10
IsoFlav_R2	6	40	0	0	0	0	5	5
JWhoisServer	6	49	0	0	44	6	0	50
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	1	5	2	0	13
NistXTS749	2	7	1	1	3	2	0	7
RiskIt	13	57	0	10	15	11	0	36
StackOverflow	4	43	0	0	5	0	0	5
UnixUsage	8	32	0	7	10	7	0	24
WordNet	8	29	0	0	22	8	1	31
Total	91	401	9	32	140	60	7	248

6.4.1 Experiment design

Schemas

For this study a set of 16 schemas were chosen, which are detailed in Table 6.1. Although there are fewer schemas used than other experiments in my thesis this ensured that the runtime of the experiment remained practically feasible. In addition, the schemas chosen vary widely in the number of tables (2 to 22), columns (3 to 67) and constraints (0 to 50), as well as containing examples of all types of supported constraints – `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `UNIQUE` and `CHECK` constraints. The schemas were also extracted from a variety of sources, with 10 being from real-world usage. For example, the JWhoisServer schema forms part of an open source server application that implements the Internet "WHOIS" protocol, while the UnixUsage schema is used in an application that logs and monitors usage of Unix commands. Other examples include RiskIt, which is part of an insurance risk modelling and premium adjustment system, and IsoFlav_R2, which is used by the U.S Department for Agriculture to report the flavonoid content of a number of foods. I therefore argue that while the set of schemas used is smaller than other experiments in my thesis, they are sufficiently diverse to produce an adequate number and variety of mutants to both test and evaluate the techniques introduced in this Chapter. This ensures the results are generalisable to a wider range of schemas, while ensuring the running time of the experiment remains practically feasible.

DBMSs

To control for any effect that the choice of DBMS may have on the results both PostgreSQL and HyperSQL were included, chosen for their architectural differences, as discussed in Section 3.3.3. This ensures that the results can be generalised to multiple DBMSs and are not an artefact of specific implementation details or performance nuances of the DBMS in use.

Methodology

To allow a thorough comparison of the techniques and approaches described in this Chapter, a number of different experimental conditions are used, each repeated 15 times to enable the calculation of means and standard deviations. This number was chosen to ensure the computational cost of running the experiment was not too great, while gathering sufficient data to control for any small random error introduced into the results by differing levels of background processes running on the machine during the experiment[1].

Firstly, for quasi-mutant detection three techniques are tested — the simple DBMS approach, the DBMS approach with SQL transactions, and the static analysis approach. The DBMS approaches are repeated for each DBMS, to expose any difference derived from the performance of the DBMS, while the static analysis approach is included only once, as the performance is independent of the DBMS in use. This gives a total of five conditions for quasi-mutant detection. For each condition, the efficiency and effectiveness is measured in terms of the time taken and the number of quasi-mutants detected, respectively, with the simple DBMS approach representing the expected result for the other techniques.

Secondly, the time taken for mutation analysis is measured for each DBMS with and without the removal of all three types of ineffective mutant present, to determine whether the time taken to identify equivalent and redundant mutants is cost-effective when compared to the time saved by the reduced number of mutants to analyse. To statistically describe the significance of any difference, the Wilcoxon Rank-Sum test is applied and complemented with the Vargha-Delaney $\hat{A}_{12}$ effect size measure, to describe

[1]The server used was not used for any other purpose during the execution of the experiment, therefore the amount of noise added to the data should be very small and controlled for by the use of average values when analysing the results.

the practical significance, if any. The effect of ineffective mutant removal is also quantified in terms of the number of equivalent, redundant and quasi-mutants detected, to better understand their prevalence and identify any possible patterns in their occurrence.

Finally, the mutation score is recorded for each DBMS with and without ineffective mutant removal, to determine how the accuracy of the score is affected by equivalent and redundant mutants. This is expressed as an increase, decrease or no change, which is tested for significance with the Wilcoxon Rank-Sum test.

Configuration

The experiment was performed using the SchemaAnalyst tool, compiled using the Java Development Kit 7 compiler and executed with the 64-bit Oracle Java 1.7 virtual machine, in an Ubuntu 12.04 environment. All trials were executed on the same server running a 3.2.0-27 GNU/Linux 64-bit kernel, with a quad-core 2.4GHz CPU, 12GB RAM and all data stored on a local disk. Experiments used PostgreSQL version 9.1.9, running on the same machine, in its default configuration and HyperSQL version 2.2.8 using the "in-memory" setting.

Threats to validity

To reduce the possible impact of different threats to the validity of the experimental results, a number of decisions were made when designing the experiment:

Misclassification of mutants It is possible that mutants may be incorrectly classified as ineffective due to implementation mistakes in the detection functions. To reduce the risk of this impacting the results, each mutant identified as either equivalent or redundant was manually reviewed. Quasi-mutants were tested by automated means by testing them against the DBMS and check they were rejected as expected. It is however possible there are further cases of equivalence or redundancy that remain undetected, which should be subject to further investigation as part of future work.

Range of schemas The prevalence of ineffective mutants depends directly on the types of constraints used in a schema. Therefore, the choice of schemas to study has a direct impact upon how cost effective the techniques described in this Chapter are.

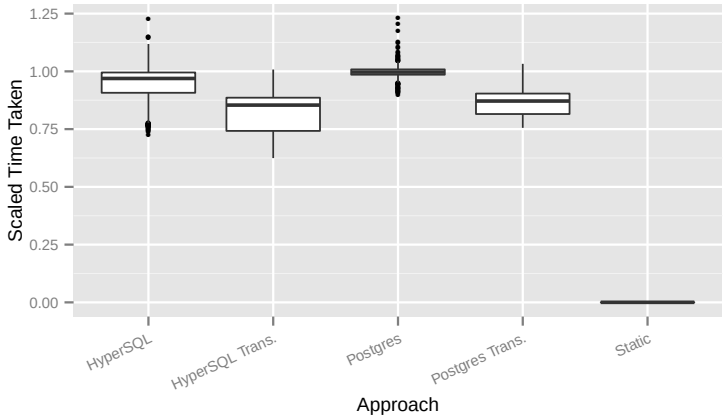


Figure 6.2: Time taken for quasi-mutant detection

The time taken when combining results from all schemas by scaling each against the maximum mean from any technique for that schema. This allows the results to be collated without being affected by the differences between schemas.

As discussed earlier in this Section, the schemas were selected to give a diverse set to study, including a number of real-world examples from open source software projects. This increases the generalisability of the experimental results.

Performance variability While the server used to perform the trials of this experiment was used exclusively for this task for its duration, differences in background tasks and caching may reduce the accuracy of the results obtained. To reduce this risk, 15 repeat trials of each configuration were performed to allow averages and standard deviation to be calculated.

6.4.2 Empirical results

Summary of results

The number of mutants produced and identified as one of the three types of ineffective mutant for each schema is shown in Table 6.2a. The same results are shown, instead grouped by the mutation operator used to generate the mutant, in Table 6.2b. In both cases, totals are provided for the number of ineffective mutants and the number of remaining, or *effective*, mutants. For each schema or operator, the savings column gives the percentage of mutants removed, calculated as:

Table 6.2: Ineffective Mutants Removed

The number of equivalent, redundant and quasi-mutants produced and removed automatically. All quasi-mutant detection techniques identified the same number of mutants, thus these result are repesentative of all three included techniques. Savings is the number of mutants removed expressed as a percentage. The savings value for the total row is calculated across the total row itself.

(a) Grouped by Schema

Schema	Produced	Equivalent	Redundant	Quasi-Mutant	∑Ineffective	∑Effective	Savings (%)
ArtistSimilarity	13	2	2	1	5	8	38.5
ArtistTerm	29	6	0	3	9	20	31.0
BankAccount	51	4	0	21	25	26	49.0
BookTown	235	22	0	0	22	213	9.4
Cloc	30	0	0	0	0	30	0.0
CoffeeOrders	115	10	0	54	64	51	55.7
Flights	70	4	3	19	26	44	37.1
IsoFlav_R2	219	0	0	0	0	219	0.0
JWhoisServer	190	12	0	0	12	178	6.3
NistDML183	40	0	2	18	20	20	50.0
NistWeather	58	3	0	23	26	32	44.8
NistXTS749	33	4	0	7	11	22	33.3
RiskIt	503	22	0	297	319	184	63.4
StackOverflow	129	0	0	0	0	129	0.0
UnixUsage	220	14	0	124	138	82	62.7
WordNet	107	20	1	0	21	86	19.6
Total	2042	123	8	567	698	1344	34.2

(b) Grouped by Operator

Operator	Produced	Equivalent	Redundant	Quasi-Mutant	∑Ineffective	∑Effective	Savings (%)
CInListElementR	4	0	0	0	0	4	0.0
CR	9	0	0	0	0	9	0.0
CRelOpE	10	0	0	0	0	10	0.0
FKColumnPairA	188	0	0	188	188	0	100.0
FKColumnPairE	287	0	5	222	227	60	79.1
FKColumnPairR	34	0	2	4	6	28	17.6
NNA	261	13	0	0	13	248	5.0
NNR	140	60	0	0	60	80	42.9
PKColumnA	327	0	0	61	61	266	18.7
PKColumnE	201	0	0	66	66	135	32.8
PKColumnR	74	0	0	21	21	53	28.4
UColumnA	427	50	0	1	51	376	11.9
UColumnE	68	0	0	2	2	66	2.9
UColumnR	12	0	1	2	3	9	25.0
Total	2042	123	8	567	698	1344	34.2

Table 6.3: Mean and standard deviation of time taken to detect quasi-mutants

Schema	Time Taken (ms)									
	HyperSQL		HyperSQL Trans.		PostgreSQL		PostgreSQL Trans.		Static	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
ArtistSimilarity	2203	117	2057	148	2811	154	2562	101	<1	<1
ArtistTerm	14852	372	13373	307	14889	331	13377	364	3	<1
BankAccount	7248	244	6362	136	7372	278	6360	177	2	<1
BookTown	227657	1934	177102	1012	238193	1495	185695	1135	<1	<1
Cloc	2361	177	2114	130	2321	145	2052	112	<1	<1
CoffeeOrders	39086	635	32108	590	39071	785	32062	561	11	<1
Flights	10113	251	9059	272	10480	211	9355	192	2	<1
IsoFlav_R2	151500	4475	137992	961	150570	1225	138145	970	<1	<1
JWhoisServer	107537	1187	96748	908	110345	1144	100450	801	<1	<1
NistDML183	3511	145	2957	113	4143	306	3445	198	2	<1
NistWeather	8643	385	7483	313	8536	198	7450	253	2	<1
NistXTS749	6633	206	6138	231	6676	213	6064	201	1	<1
RiskIt	258236	1754	210010	696	334288	1874	269025	1217	57	3
StackOverflow	34724	682	30633	510	34843	625	30796	726	<1	<1
UnixUsage	82574	1186	65771	756	102872	1341	79872	586	21	<1
WordNet	74003	852	65714	891	77998	1153	68160	529	<1	<1

Table 6.4: Mutation score with and without ineffective mutants by schema

Schema	With	Without	Difference
ArtistSimilarity	0.83	1.00	0.17
ArtistTerm	0.77	1.00	0.23
BankAccount	0.87	1.00	0.13
BookTown	0.89	0.99	0.09
Cloc	1.00	1.00	0.00
CoffeeOrders	0.84	1.00	0.16
Flights	0.88	0.95	0.07
IsoFlav_R2	1.00	1.00	0.00
JWhoisServer	0.94	1.00	0.06
NistDML183	1.00	1.00	0.00
NistWeather	0.89	0.97	0.08
NistXTS749	0.81	0.95	0.15
RiskIt	0.89	1.00	0.11
StackOverflow	0.98	0.98	0.00
UnixUsage	0.85	1.00	0.15
WordNet	0.13	0.15	0.02
Total	0.85	0.94	0.09

$$\text{Savings } (\%) = \frac{\sum \text{Ineffective}}{\text{Produced}} \times 100$$

The total rows give the sum total of mutants across all schemas or operators for each column, except the savings column, which is calculated across the total rows themselves, using the equation:

Table 6.5: Summary of mutation score change

A summary of the change to mutation scores made by removing ineffective mutants. Increase, decrease and no change show the percentage of case studies where the mutation score increase, decreased or remained the same, respectively. The "Adequate" column is the number of schemas for which the score increased from less than 1 to 1, therefore revealing a mutation adequate test suite, as percentage of the schemas where the score increased. The "Significance" column reports the statistical significance of the change in mutation score, using the Wilcoxon Rank-Sum test.

DBMS	Increase (%)	Decrease (%)	No Change (%)	Adequate (%)	Significance
HyperSQL	75	0	25	44	0.001
PostgreSQL	75	0	25	44	0.001
Both	75	0	25	44	<0.001

$$\text{Total Savings (\%)} = \frac{\sum \text{Ineffective}}{\sum \text{Produced}} \times 100$$

The time taken for each of the five quasi-mutant detection techniques is shown in Table 6.3, stated as a mean and standard deviation to account for variance across repeated trials. The static analysis technique, listed as "Static" in the table, is only included once as its efficiency is not affected by whether PostgreSQL or HyperSQL is being used. The number of quasi-mutants for each technique was identical across all schemas, repeat trials and both DBMSs, thus the results given in the quasi-mutant column of Table 6.2a are representative for all five techniques. The time taken by each technique is also shown in the box plots of Figure 6.2, where the time taken has been scaled between 0 and 1 against the maximum mean value of any technique for each schema. This allows the results for each schema to be combined, without allowing major structural differences between schemas to adversely affect the summarised data.

Table 6.6a shows the mean total time taken for mutation analysis for each DBMS, both with and without ineffective mutants, including the time to detect and remove those mutants using static analysis in the latter case. For each schema, the difference between the means is calculated and presented as both a time and percentage, which if positive indicates mutation is faster with removal of ineffective mutants than without. These results are summarised for each DBMS in Table 6.6b, where the "Both" row is a summary across all results (i.e., disregarding the DBMS used when summarising the data). Finally, Table 6.6c shows the results of statistical analysis of the difference in time taken, determining statistical significance using the Wilcoxon Rank-Sum test and effect sizes with the $\hat{A}_{12}$ statistic.

Table 6.6: Mutation analysis time with and without ineffective mutants

The mean time taken for mutation analysis, with and without ineffective mutants. A positive mean difference indicates removing ineffective mutants improves efficiency. In the summarised results, "both" is the time difference across all results, ignoring the DBMS being used. The Wilcoxon Rank-Sum test is used to test for statistical significance, where a p-value less than 0.05 is significant. The $\hat{A}_{12}$ effect size statistic is also applied, to determine whether any differences are practically significant, with values greater than 0.5 representing a time decrease and less than 0.5 representing a time increase. The effect size is described as "large" if less than 0.29 or greater than 0.71, "medium" if less than 0.36 or greater than 0.64 and "small" if less than 0.44 or greater than 0.56. Otherwise, the effect size is "none".

(a) Time taken

Schema	HyperSQL				PostgreSQL			
	Mean Time (ms)		Mean Diff.		Mean Time (ms)		Mean Diff.	
	Without	With	(ms)	(%)	Without	With	(ms)	(%)
ArtistSimilarity	185	240	55	23.05	5693	8588	2895	33.71
ArtistTerm	1178	1302	124	9.54	42668	55905	13238	23.68
BankAccount	1025	1071	47	4.35	30741	35458	4717	13.30
BookTown	26283	27001	718	2.66	1710434	1908186	197752	10.36
Cloc	1100	1027	-73	-7.15	28797	28838	41	0.14
CoffeeOrders	2979	3016	37	1.22	217683	257678	39995	15.52
Flights	2187	2161	-27	-1.24	79073	90303	11230	12.44
IsoFlav_R2	9316	8492	-824	-9.71	1032813	1029727	-3086	-0.30
JWhoisServer	9401	9104	-297	-3.26	646683	686089	39407	5.74
NistDML183	755	737	-17	-2.37	23262	24378	1116	4.58
NistWeather	1546	1541	-4	-0.29	44922	48610	3688	7.59
NistXTS749	892	928	36	3.83	26214	31125	4911	15.78
RiskIt	17555	17831	276	1.55	2117462	2434670	317208	13.03
StackOverflow	6415	6007	-408	-6.79	515815	514126	-1689	-0.33
UnixUsage	5661	5955	294	4.93	885947	1042460	156513	15.01
WordNet	3668	3851	183	4.76	115310	141511	26201	18.51

(b) Summary for DBMSs

DBMS	Time Difference (ms)		Time Difference (%)	
	Median	Mean	Median	Mean
HyperSQL	36.2	7.5	1.4	1.6
PostgreSQL	8071.0	50880.0	12.7	11.8
Both	229.9	25450.0	4.7	6.7

(c) Statistical significance of change

Schema	HyperSQL		PostgreSQL	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ArtistSimilarity	<0.01	1.00 (large)	<0.01	1.00 (large)
ArtistTerm	<0.01	1.00 (large)	<0.01	1.00 (large)
BankAccount	<0.01	1.00 (large)	<0.01	1.00 (large)
BookTown	<0.01	0.87 (large)	<0.01	1.00 (large)
Cloc	<0.01	0.00 (large)	0.61	0.54 (none)
CoffeeOrders	<0.01	0.75 (large)	<0.01	0.97 (large)
Flights	<0.01	0.15 (large)	<0.01	1.00 (large)
IsoFlav_R2	<0.01	0.00 (large)	<0.01	0.28 (large)
JWhoisServer	<0.01	0.01 (large)	<0.01	0.97 (large)
NistDML183	<0.01	0.00 (large)	<0.01	0.96 (large)
NistWeather	<0.01	0.25 (large)	<0.01	1.00 (large)
NistXTS749	<0.01	0.98 (large)	<0.01	1.00 (large)
RiskIt	<0.01	0.84 (large)	<0.01	1.00 (large)
StackOverflow	<0.01	0.00 (large)	0.01	0.30 (med)
UnixUsage	<0.01	1.00 (large)	<0.01	1.00 (large)
WordNet	<0.01	1.00 (large)	<0.01	1.00 (large)

The change in the mutation score is shown for each schema in Table 6.4 and summarised in Table 6.5, expressed as increase, decrease or no change. Where an increase occurred, "Adequate" represents the percentage of those schemas where the test suite was revealed as mutation adequate (i.e., the tests killed all remaining mutants). The statistical significance is reported using the Wilcoxon Rank-Sum test, where a value less than 0.05 is deemed significant.

Research questions

RQ1: *Does the use of static analysis improve the efficiency of quasi-mutant detection, compared to DBMS-based techniques?*

As shown by results in both Table 6.3 and Figure 6.2, the technique using static analysis to detect quasi-mutants is many times faster than either the simple DBMS technique or the SQL transaction optimised technique, regardless of whether PostgreSQL or HyperSQL is used. In the worst case, the static technique takes approximately two seconds less than the next fastest technique (<1ms compared to 2,052ms using PostgreSQL with transactions for Cloc), while being approximately three and a half minutes quicker than the next fastest technique in the best case (57ms compared to 210,010ms using HyperSQL with transactions for RiskIt). The cost of applying the static technique is also highly consistent between repeat trials, with a maximum standard deviation of 3ms, which is much lower than for the DBMS-based alternatives. In addition, as previously stated, all five quasi-mutant detection techniques identified the same number of quasi mutants, for all schemas and repeat trials, and are therefore all equally effective. As the static technique both matches the effectiveness of the most accurate technique and improves significantly on its efficiency, identifying quasi-mutants many magnitudes quicker, it is clearly the superior of the five techniques.

RQ1 Summary: *The static analysis approach for detecting quasi-mutants is significantly faster than the next best approach, often taking many magnitudes less time, whilst achieving the same results across all schemas and both DBMSs.*

RQ2: *What impact does ineffective mutant removal have on the time taken for mutation analysis?*

The results of Tables 6.6a to 6.6c show that the effect of removing ineffective mutants on the time taken for mutation analysis varies depending on the choice of schema and DBMS, ranging from a relatively small increase to large decreases.

When using HyperSQL, the difference in mean times taken for mutation analysis vary between -824 and 718ms, representing a percentage difference in mean time of -9.71 to 23.05% across all schemas. For 9 of the 16 schemas a mean time saving is achieved when removing ineffective mutants, meaning it is cost-effective to identify and discard them. In all 9 of these cases, the difference in time taken is statistically significant (p-value <0.05) and have a "large" effect size, according to the Wilcoxon Rank-Sum test and $\hat{A}_{12}$ effect size statistic, respectively. For the other 7 schemas the removal of ineffective mutants causes the mean time for mutation analysis to increase, with this increase being statistically significant and having a "large" effect size in all 7 cases. To summarise, Table 6.6b shows that when using HyperSQL and averaging across all schemas overall these time differences represent a very small time saving, with a mean of 7.5ms or 1.6%.

When using PostgreSQL, the difference in mean times with and without ineffective mutants varies between -3,086 and 317,208ms, which equates to a percentage difference of -0.33 to 33.71%. Averaging these results across all schemas gives a mean difference of 50,880ms, or 12.7%. A mean time saving is achieved by using ineffective mutant removal for 14 of the 16 schemas, with the reduction being statistically significant for all but one. Where the decrease was statistically significant the effect size of the difference was "large". For the remaining two schemas where the time for mutation analysis was increased, IsoFlav_R2 and StackOverflow, the difference was also statistically significant and with "large" and "medium" effect sizes, respectively. However, the mean increases of 3,086 and 1,689ms for these two schemas only represent increases of 0.30 and 0.33% when considered as a proportion of the total time taken, and are therefore unlikely to appreciably impact the practical usefulness of mutation analysis. In summary, the results show that when using PostgreSQL, removing ineffective mutants will reduce the time taken for mutation analysis by statistically significant degree in the majority of cases (13 out of 16 schemas in this experiment), with best case decreases of approximately one third, or in excess of 5 minutes in real-terms.

To conclude, while the choice of DBMS affects the cost-effectiveness of ineffective mutant removal as the results of Table 6.6b show, for both PostgreSQL and HyperSQL on average there is at least a small time saving, with mean values of 11.8% and 1.6%, respectively. Overall, combining results across both DBMSs gives a mean time difference of 25,540ms, or 6.7%. The large difference between DBMSs may be accounted for by the performance difference between them — as the results suggest that HyperSQL is generally

much faster than PostgreSQL, the time taken for static analysis to detect ineffective mutants will represent a larger proportion of the total time taken for mutation analysis. As the maximum increase in time taken for HyperSQL is less than 1 second this is arguably insignificant in practical terms, while the maximum decrease for PostgreSQL of over 5 minutes is significant. Therefore, the results of this experiment show that in general removing ineffective mutants is beneficial in terms of efficiency, even considering the cost of static analysis, and in many of the tested configurations leads to a practically significant reduction in time taken for mutation analysis, especially when using PostgreSQL.

RQ2 Summary: *While the removal of ineffective mutants is not always cost effective, especially when using HyperSQL, in real-terms any increase in time taken is very small. On the other hand, when removal is cost effective – often when using PostgreSQL – the time saving is practically significant, such that overall removing ineffective mutants does generally improve efficiency.*

RQ3: *Are the scores produced by mutation analysis significantly affected by ineffective mutant removal?*

The results in Table 6.5 show that whether using PostgreSQL or HyperSQL, removal of ineffective mutants caused a change in the mutant score for 75% of the 16 schemas in this experiment, with the score increasing in all of those cases. This shows that for 12 schemas the mutation score obtained with ineffective mutants present led to an under-estimate of the "true" mutation score. These increases in score were shown to be statistically significant (p-value <0.05) using the Wilcoxon Rank-Sum test.

In practical terms, these results mean that ineffective mutant removal reduced the number of mutants that would require human inspection, thus reducing the overall human-time cost of mutation analysis. Where the mutation score did change due to ineffective mutant removal, in 44% of cases the score increased to 1, meaning the test suite was revealed to be mutation adequate (i.e., it killed all of the remaining mutants). This means for those schemas, ineffective mutant removal reduced the number of mutants requiring human inspection to nil.

RQ3 Summary: *Ineffective mutant removal is able to improve the effectiveness of mutation analysis, by ensuring the mutation score is accurate by reducing the impact of ineffective mutants, as well as increase efficiency, decreasing the human costs of mutation analysis by reducing the set of mutants requiring human inspection.*

6.5 Summary

This Chapter described three types of "ineffective" mutants that negatively impact on the efficiency and effectiveness of mutation analysis for relational database schemas. Equivalent mutants behave identically to the original, non-mutant schema and therefore cannot be killed by any test case. Structurally equivalent mutants are identical to each other except for small possible differences in the SQL used to define them, while behaviourally equivalent mutants contain different constraints that accept and reject the same set of data. Redundant mutants are equivalent to one or more other mutants, either structurally or behaviourally, and thus all but one can be removed as duplicates of each other. Quasi-mutants are syntactically valid for one DBMS, but invalid for another, and may prevent the use of certain efficiency improving techniques for mutation analysis. As detecting each of these types of ineffective mutant by human inspection would be impractically expensive, this Chapter next described a series of techniques for automatically identifying them, using static analysis with a collection of patterns and detection functions. Finally, an empirical study was used to assess the impact of ineffective mutant removal on mutation analysis. Quasi-mutant detection using static analysis was compared to approaches that directly used the DBMS, include one optimised by using SQL transactions, and shown to be many magnitudes quicker. The effect of removing all types of ineffective mutants on the time taken varied according to the DBMS used, but overall led to a 6.7% decrease on average. In terms of effectiveness, the mutation score increased for 75% of schemas, with 44% of those revealing the test suite to be mutation adequate. In summary, removing ineffective mutants improves both the efficiency of mutation analysis and the accuracy of the results obtained.

Chapter 7

Improving the Efficiency of Mutation Analysis for Relational Database Schemas

7.1 Introduction

Mutation analysis provides a means of assessing the quality of a test suite, but can prove computationally expensive due to the repeated execution of potentially many tests for each mutant. This Chapter describes a collection of techniques to improve the efficiency of mutation analysis for relational database schemas, while ensuring the resulting mutation score is left unaltered. Some take inspiration from existing techniques used for program mutation (e.g., [110, 97]), implementing domain-appropriate counter-parts, while others are more specific to relational database schema mutation, which may not have corresponding equivalents in other applications.

Previously described in Section 1.3.2, the 'Original' mutation analysis technique acts as the unoptimised standard against which other techniques are compared. An algorithm for this technique is given below, followed by a discussion of several algorithms for mutation analysis of relational database schemas that incorporate a number of optimisations – mutant schemata, parallelisation and test artefact minimisation. Following

this, a "virtual" mutation analysis technique is described, which avoids costly DBMS communication by exploiting the internal model of a DBMS SchemaAnalyst uses during data generation to quickly and accurately perform mutation analysis without the use of a database. Finally, each of these techniques are evaluated with an empirical study using a wide range of schemas and multiple DBMSs, to confirm that the mutation scores for each are unaltered and quantify their impact on the time taken for mutation analysis.

To summarise, the contributions this Chapter makes are:

1. Algorithmic descriptions of a collection of novel techniques for mutation analysis of relational database schemas, including domain-specific techniques specialised for integrity constraint mutants; and

2. An empirical study to assess the accuracy of results obtained by these techniques and the affect they have on the efficiency of mutation analysis, across multiple DBMSs and a representative range of schemas.

7.2 Original Approach

To enable an empirical comparison of two data generation approaches, I proposed a technique for the mutation analysis of relational database schemas, using SQL `INSERT` statements as the test suite [62]. By recording whether each `INSERT` is accepted or rejected by database created using the schema with and without mutation, it is possible to determine whether each mutant is killed by the test suite or not, thus allow a mutation score to be calculated. The basic steps of the technique, repeated for each mutant, are as follows:

1. Create a database with the mutant schema by executing `CREATE` statements;

2. Execute each `INSERT` statement in the test suite against the database;

3. Compare the acceptance of each `INSERT` statement to the acceptance when using the non-mutated schema. If there are any differences the mutant is killed; and

4. Remove the database from the DBMS with `DROP` statements.

Algorithm 19 "Original" mutation analysis technique (as described by Kapfhammer et al. [62])

```
Killed ← ∅
for mutant do
    Create tables in database for mutant
    for insert in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        mutantResult ← executeWithDBMS(insert)
        if originalResult ≠ mutantResult then
            Killed ← Killed ∪ {mutant}
    Remove tables in database for mutant
```

Once this process has been repeated for each mutant, a mutation score can be calculated for the test suite using the equation:

$$Mutation\ score = \frac{\|Killed\ mutants\|}{\|Mutants\|}$$

This approach, referred to as the "Original" technique, is described more formally in Algorithm 19. As no optimisations are employed in this technique it can become very computationally expensive, especially for large schemas, as shown later in Section 7.6. Sections 7.3 and 7.4 explore a number of possible improvements to this mutation analysis algorithm, while Section 7.5 investigates an alternative approach leveraging the fitness functions used in the search-based data generation process of SchemaAnalyst to avoid the need for DBMS interaction.

7.3 Schemata Techniques

As previously described in Section 2.3.5, the *mutant schemata* approach to mutation analysis produces a "meta-mutant" that contains both the original artefact and its mutants. In program mutation, this meta-mutant contains a conditional branch for each mutation, where either the original or mutant code is executed based upon which mutant is enabled at runtime. This aims to reduce the time needed for mutation analysis by producing only a single version of the program that needs to be compiled and executed, rather than one per mutant. Although the meta-mutant program itself will be much larger than the original, and thus more expensive to compile, each extra mutant only incurs an additional conditional branch, rather than requiring a recompilation of the whole program.

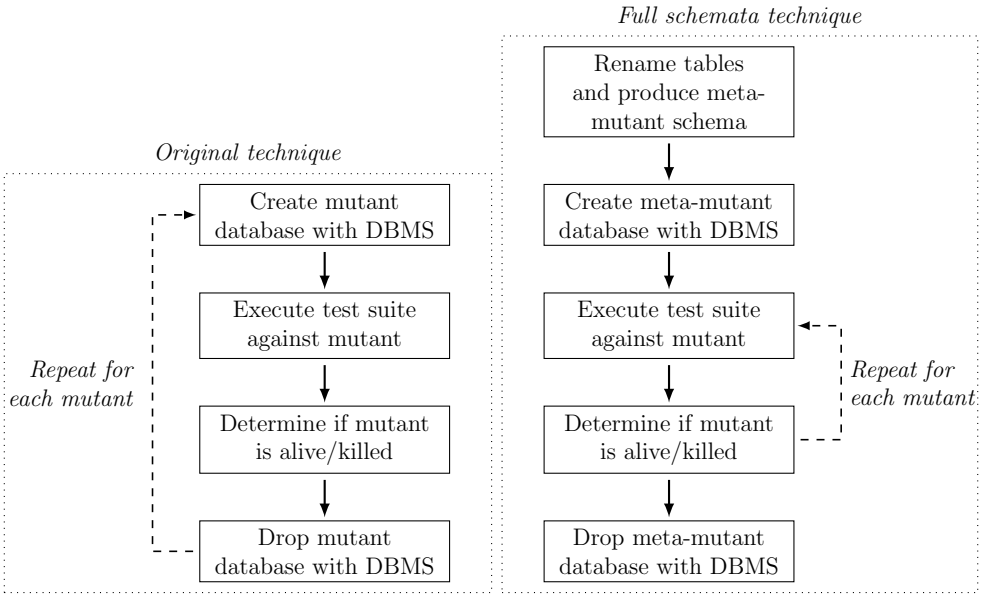


Figure 7.1: A comparison of the Original and Full schemata techniques, showing the reduced number of steps which are repeated per mutant by the latter.

The rest of this Section describes three techniques – *Full schemata*, *Minimal schemata* and *Minimal*$^+$ *schemata* – that incorporate the concepts of mutant schemata in the context of mutation of relational database schemas, with the aim of reducing the time taken. The impact each has on the efficiency of mutation analysis is evaluated later in Section 7.6.

7.3.1 Full schemata

The *Full schemata* technique produces a single relational database schema that contains both the schema under test and all of its mutants. This reduces the number of database interactions taken to execute `CREATE` and `DROP` statements, from two per mutant to two times overall, thus reducing the time taken communicating with the DBMS.

Figure 7.1 gives an overview of the major steps used by this technique compared to the Original technique, showing the reduced number of steps that must be repeated for each mutant, while Algorithm 20 more formally describes this technique.

Algorithm 20 Full schemata mutation analysis technique

```
▷ 1. Meta-mutant creation
for each mutant do
    Prefix names of tables in mutant with unique mutant ID
    Add tables of mutant to metamutant
Create tables in database for metamutant

▷ 2. Mutant evaluation
Killed ← ∅
for mutant do
    for insert in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        insert′ ← insert modified to use mutant ID for table names
        mutantResult ← executeWithDBMS(insert′)
        if originalResult ≠ mutantResult then
            Killed ← Killed ∪ {mutant}

▷ 3. Clean up
Remove tables in database for metamutant
```

Although this technique reduces the repeated part of the algorithm, in exchange the size of the database created will be increased, due to the extra tables included in the schema. As creating a single schema involves including an extra copy of each table for every mutant, this technique must also include an extra "renaming" step in which table names are prefixed with a unique identifier (e.g., `m1_name`, `m2_name`, etc...), to avoid identifier collisions, before being merged into a single schema. For example, given the schema, `s` and mutants, $\texttt{m}_1$ and $\texttt{m}_2$:

```
s:   x (a INT, b INT)
     y (c INT)
m1:  x (a INT NOT NULL, b INT)
     y (c INT)
m2:  x (a INT, b INT NOT NULL)
     y (c INT)
```

...then the Original technique would communicate with the DBMS six separate times to create and drop the database for use in mutation analysis:

	Statement:	Description:
1.	`CREATE TABLE x (a INT, b INT)`	*Create* s
	`CREATE TABLE y (c INT)`	
2.	`DROP TABLE x`	*Drop* s
	`DROP TABLE y`	
3.	`CREATE TABLE x (a INT NOT NULL, b INT)`	*Create* m_1
	`CREATE TABLE y (c INT)`	
4.	`DROP TABLE x`	*Drop* m_1
	`DROP TABLE y`	
5.	`CREATE TABLE x (a INT, b INT NOT NULL)`	*Create* m_2
	`CREATE TABLE y (c INT)`	
6.	`DROP TABLE x`	*Drop* m_2
	`DROP TABLE y`	

However, the Full schemata technique instead produces a meta-mutant schema, $\texttt{s}_{\texttt{meta}}$ that contains all of the tables of $\texttt{s}$, $\texttt{m}_1$ and $\texttt{m}_2$, with mutant tables prefixed to avoid name collisions:

$\texttt{s}_{\texttt{meta}}$:
```
x (a INT, b INT)
y (c INT)
m1_x (a INT NOT NULL, b INT)
m1_y (c INT)
m2_x (a INT, b INT NOT NULL)
m2_y (c INT)
```

...which then reduces the number of database interactions, to once to execute the appended `CREATE` statements and once for the `DROP` statements:

	Statement:	Description:
1.	`CREATE TABLE x (a INT, b INT)`	*Create* s_{meta}
	`CREATE TABLE y (c INT)`	
	`CREATE TABLE m1_x (a INT NOT NULL, b INT)`	
	`CREATE TABLE m1_y (c INT)`	
	`CREATE TABLE m2_x (a INT, b INT NOT NULL)`	
	`CREATE TABLE m2_y (c INT)`	
2.	`DROP TABLE x`	*Drop* s_{meta}
	`DROP TABLE y`	
	`DROP TABLE m1_x`	
	`DROP TABLE m1_y`	
	`DROP TABLE m2_x`	
	`DROP TABLE m2_y`	

During mutation analysis, it is then possible to test with any chosen mutant by simply prefixing the table names used in each `INSERT` statement with the relevant mutant identifier (e.g., `m1_`, `m2_`, etc...).

7.3.2 Minimal schemata

As with the Full schemata technique, the *Minimal schemata* produces a meta-mutant containing both the schema under test and its mutants. However, for each mutant meta-mutant created by the Minimal schemata technique only includes the table altered by the mutation – referred to as the *affected* table – rather than one copy of all tables. Compared to Full schemata, this vastly reduces the number of tables in the meta-mutant schema from "$tables \times (mutants + 1)$" to "$tables + mutants$". For example, given the previous example schema, `s`, and mutants, $\texttt{m}_1$ and $\texttt{m}_2$:

```
s:   x (a INT, b INT)
     y (c INT)
m1:  x (a INT NOT NULL, b INT)
     y (c INT)
m2:  x (a INT, b INT NOT NULL)
     y (c INT)
```

...while the meta-mutant for Full schemata contains a total of six tables, the one produced by Minimal schemata contains only four – only including the affected `x` table for mutants $\texttt{m}_1$ and $\texttt{m}_2$ because the `y` table was unaltered by their mutations:

Full schemata:

```
s_meta:  x (a INT, b INT)
         y (c INT)
         m1_x (a INT NOT NULL, b INT)
         m1_y (c INT)
         m2_x (a INT, b INT NOT NULL)
         m2_y (c INT)
```

Minimal schemata:

```
s_meta:  x (a INT, b INT)
         y (c INT)
         m1_x (a INT NOT NULL, b INT)
         m2_x (a INT, b INT NOT NULL)
```

Because there may be `FOREIGN KEY` constraints in the schema under test, it is possible for the affected table of a mutant to contain a reference to a table not contained in the meta-mutant. If left unresolved this would cause whichever DBMS is being used to reject the `CREATE` statements as syntactically invalid, preventing mutation analysis from continuing. To avoid this, the Minimal schemata technique remaps the referenced tables in all `FOREIGN KEY` constraints to the non-mutated copy in the schema under test, which will lead to the same results under the assumption that only a single mutation is made (i.e., this cannot produce correct results in a higher-order mutation context). For example, given the schema, `s`, which contains a `FOREIGN KEY` constraint, and its mutants $\texttt{m}_1$ and $\texttt{m}_2$:

```
s:   x (a INT, FOREIGN KEY a REFERENCES y(b))
     y (b INT)
m1:  x (a INT NOT NULL, FOREIGN KEY a REFERENCES y(b))
     y (b INT)
m2:  x (a INT, FOREIGN KEY a REFERENCES y(b))
     y (b INT NOT NULL)
```

...the Full schemata and Minimal schemata techniques would produce two differing meta-mutants:

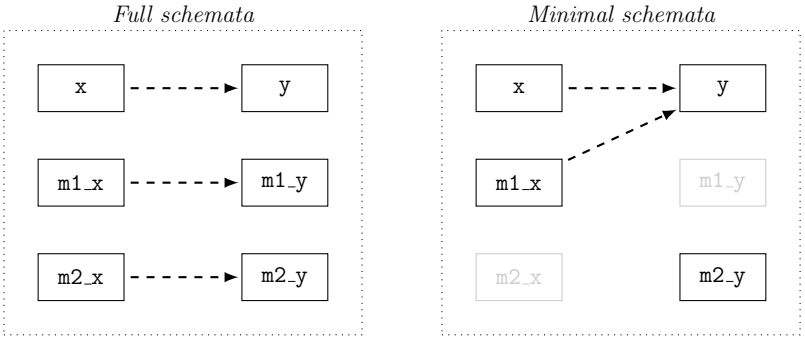
Full schemata:

```
smeta: x (a INT, FOREIGN KEY a REFERENCES y(b))
       y (b INT)
       m1_x (a INT NOT NULL,
            FOREIGN KEY a REFERENCES m1_y(b))
       m1_y (b INT)
       m2_x (a INT, FOREIGN KEY a REFERENCES m2_y(b))
       m2_y (b INT NOT NULL)
```

Minimal schemata:

```
smeta: x (a INT, FOREIGN KEY a REFERENCES y(b))
       y (b INT)
       m1_x (a INT NOT NULL,
             FOREIGN KEY a REFERENCES y(b))
       m2_y (b INT NOT NULL)
```

In the meta-mutant produced by the Full schemata technique, each of the `FOREIGN KEY` constraints is updated to refer to the copy of the table prefixed with the mutant ID (e.g., $\texttt{m}_1$ and $\texttt{m}_2$). However, both the `FOREIGN KEY` constraints meta-mutant from the Minimal schemata technique reference the same table, `y`. Note also in the case of $\texttt{m}_2$, there is no `FOREIGN KEY` constraint to remap, because `y` is the affected table, not `x`. Continuing with the above example, the differing dependencies between tables for Full schemata and Minimal schemata, where an arrow from a to b means a depends on b, can be visualised as:



As well as reducing the size of the meta-mutant, Minimal schemata aims to reduce the cost of test suite execution. Rather than repeatedly running all tests for every mutant,

the Minimal schemata technique only executes those `INSERT` statements that attempt to add data to the affected table. Because `FOREIGN KEY` constraints are mapped to the same "shared" copy of the reference table, the data required to satisfy these constraints can be added once and referred to by each of the mutants. Naively, this could be achieved by modifying the `INSERT` statements of each test at execution time, to refer to either the desired mutant table or shared reference table, as needed, and ignoring any other test data. However, this required insertion and deletion of data into the shared reference table to be repeated for each mutant. Instead, the Minimal schemata technique only inserts the shared data once, interleaving this with executing statements for each mutant table as necessary according to the test data. This proceeds according to the following steps for each `INSERT` statement:

1. Execute the statement with the non-mutated copy of the table named in the `INSERT`, to determine the acceptance status of the schema under test and in case referenced by a `FOREIGN KEY` constraint.

2. Execute the statement for each mutated copy of the table named in the `INSERT` that exists in the meta-mutant, marking the mutant as killed if the acceptance differs from the schema under test.

For example, given the schema $\mathtt{s}$, mutants $\mathtt{m}_1$ and $\mathtt{m}_2$, meta-mutant $\mathtt{s}_{\mathtt{meta}}$, and test sequence $\mathtt{t}_1$ to $\mathtt{t}_2$:

```
s:   x (a INT REFERENCES y(b))
     y (b INT)
m1:  x (a INT NOT NULL REFERENCES y(b))
     y (b INT)
m2:  x (a INT REFERENCES y(b))
     y (b INT NOT NULL)
```

```
smeta: x (a INT REFERENCES y(b))
       y (b INT)
       m1_x (a INT NOT NULL,
             REFERENCES y(b))
       m2_y (b INT NOT NULL)
```

```
t1:  y(b=1)
t2:  x(a=1)
```

...then the `INSERT` execution ordering used by Minimal schemata would be as follows:

1. Execute $\mathtt{t}_1$ with table `y`.

2. Execute $\mathtt{t}_1$ with table `m2_y`.

3. Execute $\mathtt{t}_2$ with table `x`.

4. Execute $\mathtt{t}_2$ with table `m1_x`.

Algorithm 21 Minimal schemata mutation analysis technique

```
▷ 1. Meta-mutant creation
for each mutant do
    mutant′ ← mutant with non-affected tables removed
    Prefix names of tables in mutant′ with unique mutant ID
    Add tables of mutant′ to metamutant
Create tables in database for metamutant

▷ 2. Mutant evaluation
Killed ← ∅
for each insert in testSuite do
    originalResult ← Pre-computed result of insert with non-mutant
    affectedTable ← Table insert is involving
    affectedMutants ← Mutants that mutated affectedTable
    executeWithDBMS(insert)                    ▷ (To satisfy FOREIGN KEY references)
    for each affectedMutant do
        insert′ ← insert modified to use mutant ID for table names
        mutantResult ← executeWithDBMS(insert′)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}

▷ 3. Clean up
Remove tables in database for metamutant
```

This process ensures that for any `FOREIGN KEY`, the expected data is present in the reference table, without requiring the `INSERT` statements to be executed multiple times, for each mutant.

If a mutant is killed, no more tests are executed against that mutant. However, if a mutant returns the same results as the schema under test for all `INSERT` statements executed against it, then the test suite is unable to detect the mutant and it is alive. The overall algorithm for the Minimal schemata technique is shown in Algorithm 21.

7.3.3 Minimal$^+$ schemata

The *Minimal$^+$ schemata* technique aims to improve upon Minimal schemata by further reducing the size of the meta-mutant. Where Minimal schemata includes a full copy of the affected table for each mutant, Minimal$^+$ schemata only includes the mutated constraint in the copied table, reducing the overall number of constraint in the meta-mutant schema. For example, given the schema, `s`, and mutants, $\texttt{m}_1$ (`NOT NULL` added to `b` in table `x`) and $\texttt{m}_2$ (`NOT NULL` added to `c` in table `y`):

```
s:   x (a INT NOT NULL, b INT)
     y (c INT, d INT PRIMARY KEY)
m1:  x (a INT NOT NULL, b INT NOT NULL)
     y (c INT, d INT PRIMARY KEY)
m2:  x (a INT NOT NULL, b INT)
     y (c INT NOT NULL, d INT PRIMARY KEY)
```

...then the Minimal schemata and Minimal$^+$ schemata techniques would produce the following meta-mutant schemas:

Minimal schemata:

```
smeta: x (a INT NOT NULL, b INT)
       y (c INT, d INT PRIMARY KEY)
       m1_x (a INT NOT NULL, b INT NOT NULL)
       m2_y (c INT NOT NULL, d INT PRIMARY KEY)
```

Minimal$^+$ schemata:

```
smeta: x (a INT NOT NULL, b INT)
       y (c INT, d INT PRIMARY KEY)
       m1_x (a INT, b INT NOT NULL)
       m2_y (c INT NOT NULL, d INT)
```

Notably, while both techniques contain the same number of tables, reduced by 1/3 from 6 to 4 when compared to Full schemata, in the meta-mutant produced by Minimal$^+$ schemata only includes the mutated constraints in tables `m1_x` and `m2_y`. This means that for each `INSERT` statement executed for a mutant there are fewer constraints that must be checked by the DBMS, which should in turn reduce the time taken to execute each statement.

However, the applicability of this technique depends upon the type of modification made by the mutation operator. An operator either increases or decreases the range of `INSERT` statements that can be accepted into a database created with the schema. As a consequence, detection of a mutant may either be by an `INSERT` being rejected by the non-mutant and accepted by the mutant, for *Addition* type operators, or vice-versa, for *Removal* or *Exchange* type operators. For example, for the schema, `s`, and mutant produced by removal operator, m_1 (`PRIMARY KEY` removed by the PKColumnR operator):

```
s:   x (a INT PRIMARY KEY)
m1:  x (a INT)
```

Then a sequence of `INSERT` statements such as the following list would be required "kill" the mutant, by being rejected by the non-mutant and accepted by the mutant, because the mutant has fewer constraints on the accepted data:

	Acceptance by...	
	Original	*Mutant*
1. `INSERT INTO TABLE x VALUES (1)`	✓	✓
2. `INSERT INTO TABLE x VALUES (1)`	✗	✓

Algorithm 22 Minimal$^+$ schemata mutation analysis technique

```
▷ 1. Mutant division and delegated mutation analysis
mutants_RE ← Mutants produced by removal or exchange operators
mutant_A ← Mutants produced by addition operators
Killed ← MINIMALSCHEMATA(mutants_RE, testSuite)

▷ 2. Meta-mutant creation
for each mutant in mutants_A do
    mutant′ ← mutant with non-affected tables and non-affected constraints removed
    Prefix names of tables in mutant′ with unique mutant ID
    Add tables of mutant′ to metamutant
Create tables in database for metamutant

▷ 3. Mutant evaluation
for each insert in testSuite do
    originalResult ← Pre-computed result of insert with non-mutant
    affectedTable ← Table insert is involving
    affectedMutants ← Mutants that mutated affectedTable
    executeWithDBMS(insert)                    ▷ (To satisfy FOREIGN KEY references)
    for each affectedMutant do
        insert′ ← insert modified to use mutant ID for table names
        mutantResult ← executeWithDBMS(insert′)
        if originalResult ≠ mutantResult then
            K ← K ∪ {mutant}

▷ 4. Clean up
Remove tables in database for metamutant
```

However, if the mutant is produced with an addition operator, such as with `s` and $\texttt{m}_1$ (`PRIMARY KEY` added with the PKColumnA) as follows:

```
s:   x (a INT)
m1:  x (a INT PRIMARY KEY)
```

Then to kill the mutant, it is necessary to produce a sequence of `INSERT` statements that are accepted by the non-mutant and rejected by the mutant, as the mutation has increased the constraints on the range of accepted data, for example:

	Acceptance by...	
	Original	*Mutant*
1. `INSERT INTO TABLE x VALUES (1)`	✓	✓
2. `INSERT INTO TABLE x VALUES (1)`	✓	✗

Because the Minimal$^+$ schemata technique removes all but the mutated constraint, it cannot reliably infer whether a statement rejected by the non-mutant but accepted

by the meta-mutant representation of a particular mutant is due to the mutation or because of the constraints removed when producing the meta-mutant. However, if an `INSERT` statement is accepted by the non-mutant but rejected by the mutant table in the meta-mutant, it must be due to the mutation, as the mutated constraint is the only constraint present. Therefore, the extra optimisation of Minimal$^+$ schemata can be used for mutants produced by addition operators, but not for those generated by removal or exchange operators. As a consequence, prior to mutation analysis the Minimal$^+$ schemata technique divides the mutants of the schema under test according to the modification type of the operator used to generate them, delegating to the Minimal schemata technique to obtain a mutation score for those mutants it cannot accurately analyse itself and combining this with its own results to give a final mutation score. Algorithm 22 shows the overall mutation analysis approach of the Minimal$^+$ schemata technique.

7.4 Parallelisation Techniques

Parallel algorithms aim to exploit the presence of multi-core processors, even on commodity hardware, to reduce their overall running time by dividing the problem into multiple parts that can be executed concurrently and then recombining the results. Distributing the units of work across multiple separate computers, such as with a computer cluster or grid computer, may also be described as parallel computation, however for this Section I only refer to the former type of parallelism.

Section 2.3.5 describes a number of existing approaches to use parallel execution to reduce the cost of mutation analysis for testing programs. This Section discusses two novel techniques for applying parallelisation to mutation analysis of relational database schemas – *Just-in-Time schemata* and *Up-Front schemata* . These are later evaluated in Section 7.6 to determine what effect parallel algorithms have on the time taken for mutation analysis.

7.4.1 Just-in-Time schemata

The *Just-in-Time schemata* technique can be thought of as a parallel version of the Original technique, with `CREATE` and `DROP` statements being executed for each mutant

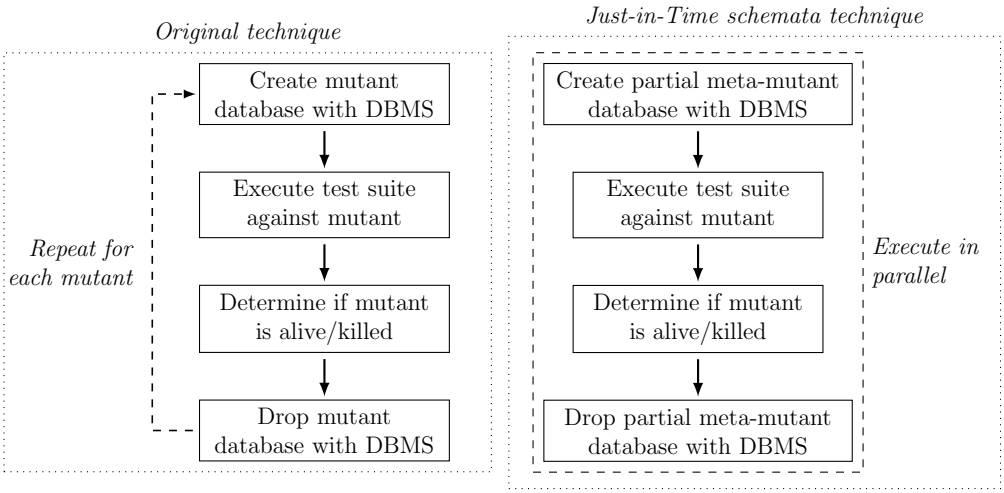


Figure 7.2: A comparison of the Original and Just-in-Time schemata techniques, showing the parallelisation of the loop in the latter case.

Algorithm 23 Just-in-Time schemata mutation analysis technique

```
▷ 1. Mutant renaming
for each mutant do
    Prefix names of tables in mutant with unique mutant ID

▷ 2. Mutant evaluation
Killed ← ∅
parallel for mutant do
    Create tables in database for mutant
    for insert in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        mutantResult ← executeWithDBMS(insert)
        if originalResult ≠ mutantResult then
            Killed ← Killed ∪ {mutant}
    Remove tables in database for mutant
```

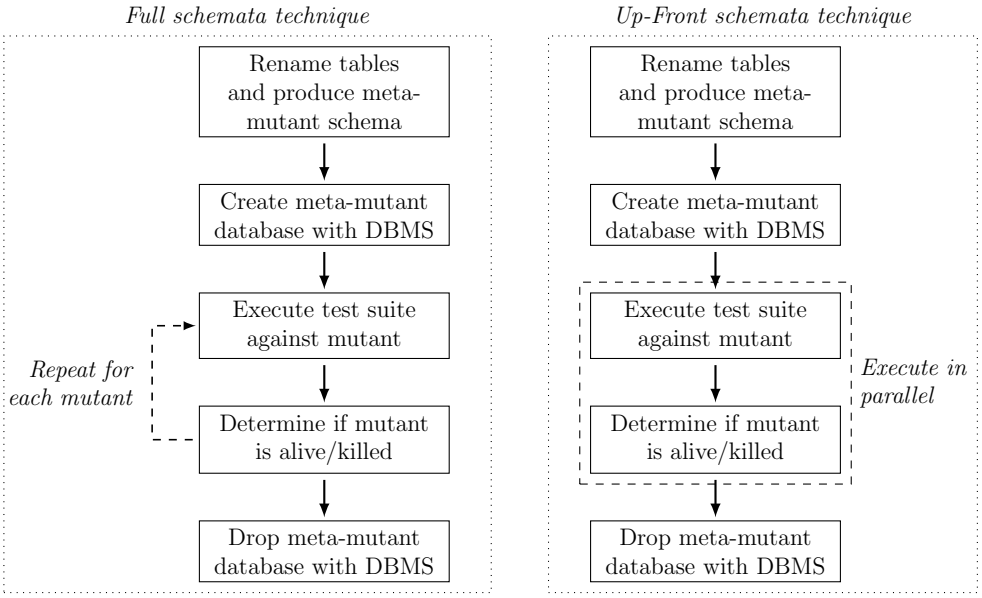


Figure 7.3: A comparison of the Full schemata and Up-Front schemata techniques, showing the use of parallel computation by the latter.

just prior to and after analysis, respectively, hence former part of its name. However, due to the parallel analysis of mutants it is very likely that table name collisions will be caused within the DBMS, with analysis of two or more mutants attempting to simultaneously store tables of the same name in the database. To prevent this, the tables are renamed according to the prefixing strategy of the schemata techniques (i.e., `m1_`, `m2_`, etc...), such that a partial meta-mutant is used for each mutant. Therefore, Just-in-Time schemata is both a schemata and parallel technique.

Figure 7.2 shows a comparison of this technique compared to the Original technique. Over the course of mutation analysis, the tables added will be identical to the meta-mutant produced by Full schemata, however only part of the meta-mutant will be present at any given moment, depending on how many mutants are being analysed simultaneously. The overall approach used for the Just-in-Time schemata technique is shown in Algorithm 23.

Algorithm 24 Up-Front schemata mutation analysis technique

```
▷ 1. Meta-mutant creation
for each mutant do
    Prefix names of tables in mutant with unique mutant ID
    Add tables of mutant to metamutant
Create tables in database for metamutant

▷ 2. Mutant evaluation
Killed ← ∅
parallel for mutant do
    for insert in testSuite do
        originalResult ← Pre-computed result of insert with non-mutant
        insert′ ← insert modified to use mutant ID for table names
        mutantResult ← executeWithDBMS(insert′)
        if originalResult ≠ mutantResult then
            Killed ← Killed ∪ {mutant}

▷ 3. Clean up
Remove tables in database for metamutant
```

7.4.2 Up-Front schemata

While Just-in-Time schemata can be likened to a parallel version of the Original technique, *Up-Front schemata* is a parallel equivalent to the Full schemata technique. As with Full schemata a meta-mutant is produced containing all mutants, which includes a full copy of all tables for each mutant. This is used both prior to mutation analysis, to create a database, and afterwards, to remove the database, whilst requiring only one database interaction for each. However, the analysis of each mutant can then be performed in parallel, rather than serial. Figure 7.3 shows this conceptual difference between Full schemata and Up-Front schemata, while Algorithm 24 shows the approach used by the technique.

7.5 Virtual Mutation Analysis

Communication between different processes can prove to be very computationally expensive, even when those processes are running on the same physical machine (e.g., via Unix domain sockets), compared to sharing data between different parts of the same process

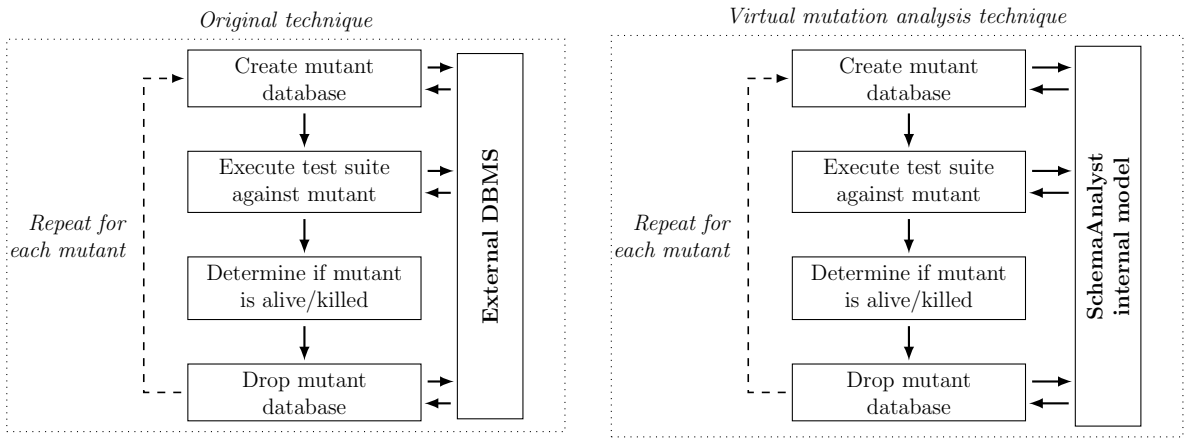


Figure 7.4: A comparison of the Original and Virtual mutation analysis techniques, showing the similarity of the algorithms except for the replacement of the external DBMS with an internal model of DBMS behaviour.

(either with single or multiple threads of execution). All of the previously described mutation analysis techniques require interaction with a database instance hosted in a DBMS, and therefore incur such a performance cost. The *Virtual* mutation analysis technique aims to reduce the time taken for mutation analysis by removing this requirement, by instead exploiting a model of DBMS behaviour used elsewhere in SchemaAnalyst to perform mutation analysis without creating a database.

As discussed in Section 3.5, the data generation component of SchemaAnalyst is able to automatically generate test suites of data that exercise the constraints of a relational database schema, according to a chosen coverage criterion. To generate the required data a search-based technique is used, with SchemaAnalyst employing the AVM algorithm, as described in Section 3.5.2. The search process is guided according to a fitness function that evaluates whether a given set of data tests the constraints as required, either by satisfying or violating all or some of them. This function returns a score between 0 and 1 that can be used to determine how far away a set of data is from meeting the current requirements, such that it can direct the search towards a set of data values that will exhibit the correct response from the DBMS. In this way, the fitness functions produced during data generation model the expected behaviour of the DBMS.

The Virtual mutation analysis technique exploits these fitness functions to determine whether a test case for a schema will be accepted or rejected by a given DBMS. This is

compared to the Original technique in Figure 7.4. By repeating this for all cases in a test suite, repeating the process for each mutant of a schema, it is possible to perform mutation analysis without any DBMS interaction. As the fitness functions are designed to be quick to evaluate – as they may be evaluated many thousands of times during data generation – the Virtual mutation analysis technique is expected to reduce the time taken for mutation analysis considerably, compared to those communicating with a DBMS.

A drawback of the Virtual mutation analysis technique is that it requires the internal modelling of each DBMS within SchemaAnalyst to be wholly accurate with respect to the SQL features used in the schema under test. This means that supporting a new DBMS requires the formation of the fitness function to be adjusted according to any subtle differences in its implementation of SQL. However, in general the differences between DBMSs are relatively small, especially for the small set of SQL features used most commonly in many relational database schemas (such as those described in Section 3.4), therefore this human cost is relatively low. The correctness of this modelling is now assured, to a reasonable degree of certainty, by empirically comparing the mutation scores it achieves for a large number of schema to those produced by the other techniques in Section 7.6. Further discussion of this technique has been published elsewhere [77].

7.6 Empirical Experiment

This Section details the design and results of an empirical experiment to determine what impact the schemata, parallel and virtual mutation techniques detailed in Sections 7.3 to 7.5 have on the time taken for mutation analysis, and how this is affected by the schema under test and DBMS in use.

7.6.1 Experiment design

Schemas

Given that the number of tables, columns and constraints may significantly impact upon the performance of each the mutation analysis techniques, a total of 32 schemas were selected for this experiment. These vary from 1 to 42 tables, 3 to 309 columns and 0

Table 7.1: Schemas analysed in the empirical study

Schema	Tables	Columns	Checks	Foreign Keys	Not Nulls	Primary Keys	Uniques	∑Constraints
ArtistSimilarity	2	3	0	2	0	1	0	3
ArtistTerm	5	7	0	4	0	3	0	7
BankAccount	2	9	0	1	5	2	0	8
BookTown	22	67	2	0	15	11	0	28
BrowserCookies	2	13	2	1	4	2	1	10
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
CustomerOrder	7	32	1	7	27	7	0	42
DellStore	8	52	0	0	39	0	0	39
Employee	1	7	3	0	0	1	0	4
Examination	2	21	6	1	0	2	0	9
Flights	2	13	1	1	6	2	0	10
FrenchTowns	3	14	0	2	13	0	9	24
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
iTrust	42	309	8	1	88	37	0	134
JWhoisServer	6	49	0	0	44	6	0	50
MozillaExtensions	6	51	0	0	0	2	5	7
MozillaPermissions	1	8	0	0	0	1	0	1
NistDML181	2	7	0	1	0	1	0	2
NistDML182	2	32	0	1	0	1	0	2
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	1	5	2	0	13
NistXTS748	1	3	1	0	1	0	1	3
NistXTS749	2	7	1	1	3	2	0	7
Person	1	5	1	0	5	1	0	7
Products	3	9	4	2	5	3	0	14
RiskIt	13	57	0	10	15	11	0	36
StackOverflow	4	43	0	0	5	0	0	5
StudentResidence	2	6	3	1	2	2	0	8
UnixUsage	8	32	0	7	10	7	0	24
Usda	10	67	0	0	31	0	0	31
Total	172	975	38	49	335	114	18	554

and 134 constraints, with at least one of each of the types of constraint supported by SchemaAnalyst (`PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `UNIQUE` and `CHECK` constraints). Table 7.1 provides a detailed breakdown of the schema attributes. Many of these schemas are taken from real-world sources, as previously discussed in Section 6.4.1, thus enabling the generalisation of results from this experiment to analysis of other real-world schemas.

DBMSs

To determine whether the efficiency of any technique is affected by the choice of DBMS used this experiment included all three DBMSs supported by SchemaAnalyst– PostgreSQL, SQLite and HyperSQL. Performance differences may be caused by the differing architecture of these DBMSs — PostgreSQL operates as a standalone database server,

while SQLite and HyperSQL operate as "embedded" databases integrated into an application — and their use of available storage hardware — SQLite and HyperSQL have been configured to use an "in-memory" mode that disables writing to the slower hard disk drive, reducing the time taken waiting for input/output operations, whereas PostgreSQL may make periodic read/write operations to the hard disk drive.

Methodology

The mutation analysis techniques described in Sections 7.3 to 7.5 of this Chapter are evaluated in terms of both accuracy and efficiency, according to two metrics. These metrics were monitored over 30 repeated trials for each technique, DBMS and schema combination to ensure the results were consistent and representative, such as ensuring varying levels of background processes on the server used did not adversely affect the timing data obtained.

Firstly, to determine the accuracy of each technique, the mutation score obtained for each combination of DBMS and schema was recorded and compared to that obtained by the Original technique, which is used as the "gold standard" for score correctness. This ensures that the implementation of the technique is correct and does not contain any bugs, which might otherwise cause an incorrect result to be obtained.

Secondly, the time taken for the mutation analysis technique to return a mutation score was recorded. This excluded the time taken to generate both the mutants and test data used, to ensure that the timing data obtained represented only the time taken by the technique in use, to prevent any variable performance in other parts of the overall mutation analysis process reducing the accuracy of the experimental analysis.

Configuration

The experiment was performed using the SchemaAnalyst tool, compiled using the Java Development Kit 7 compiler and executed with the 64-bit Oracle Java 1.7 virtual machine, in an Ubuntu 12.04 environment. All trials were executed on the same server running a 3.2.0-27 GNU/Linux 64-bit kernel, with a quad-core 2.4GHz CPU, 12GB RAM and all data stored on a local disk. Experiments used PostgreSQL version 9.1.9, running on the same machine, in its default configuration, HyperSQL version 2.2.8 using the "in-memory" setting and SQLite version 3.8.2 using the "in-memory" setting.

Threats to validity

To ensure that the results obtained are both accurate and can be generalised to a broader set of relational database schemas, a range of steps were taken to mitigate the threats to their validity. A number of these are discussed below:

Original technique accuracy Given that the Original mutation analysis technique is being used as the "gold standard" to measure the correctness of the other techniques, it is essential that it itself correct. This risk is firstly minimised as a consequence of the simplicity of its implementation – that is, because the technique is relatively simple, communicating with the DBMS directly with no optimisations, there is less scope for faults to occur. Secondly, the technique has been used for the experiment in the previous Chapter as well as a number of publications with no anomalous results being identified, and has been subject to thorough manual testing.

Execution environment variability It is possible that different optimisations will perform more or less efficiently, depending on the hardware it is executed using. As previously discussed, all trials were executed on the same server, ensuring all techniques were given the same computational resources, which are sufficient to ensure that constraints such as available memory do not impact upon the results.

Range of schemas The performance of each technique may reply upon the size of a schema and the type of constraints it contains, which in turn impacts upon the number of mutants generated. It is therefore important to include a variety of schemas. As discussed earlier in this Section, many of the schemas were taken from real-world applications and were selected to give a very wide range of their attributes, such as the number of tables, columns and each type of constraint.

7.6.2 Empirical results

Summary of results

The mutation scores produced by each of the six novel techniques — Full schemata, Up-Front schemata, Just-in-Time schemata, Minimal schemata, Minimal$^+$ schemata and

Virtual mutation analysis — matched exactly with those produced by the Original technique for all DBMS and schema combinations, as well as all 30 repeat trials, and therefore are not otherwise presented in this Section.

The box plots in Figures 7.5 to 7.7 show the time taken for each mutation analysis technique and schema, where lower is better, for HyperSQL, PostgreSQL and SQLite, in turn. The boxes in the plots span from the 1st to 3rd quartile, with the line across the box marking the median value, and the whiskers extend to the furthest data point within 1.5× the inter-quartile range. Any data points outside of this are outliers, marked as filled circles.

The mean time in milliseconds taken for mutation analysis with each of the techniques for each DBMS and schema is shown in Table 7.2, where lower values are better. In each row, the fastest technique for that DBMS and schema is highlighted. Table 7.3 presents the same results but with the time taken for each technique scaled compared to the time taken by the Original technique (i.e., a result of 0.5 means taking half as long as Original, while 2.0 means taking twice as long), such that lower values are better. This is summarised across all schemas using a mean in Table 7.4.

Research questions

RQ1: *Are the mutation scores obtained by the novel analysis techniques as accurate as those from the Original technique?*

As described in the Summary of results, the mutation scores recorded for each of the novel mutation analysis techniques — Full schemata, Just-in-Time schemata, Up-Front schemata, Minimal schemata, Minimal$^+$ schemata and Virtual mutation — were consistent with those from the Original technique, for all DBMSs and schemas, across all 30 repeat trials. This provides assurance that the implementation of the six techniques is correct and free from any significant bugs, as well as allowing them to be compared solely by their efficiency, as they are exactly as effective as the "gold standard" Original technique.

RQ1 Summary: *Yes, all of the novel techniques consistently return the expected mutation scores.*

RQ2: *Do the novel techniques perform more efficiently than the Original technique, which is the most efficient technique and what impact does the choice of DBMS have on the time taken?*

Full schemata: For both HyperSQL and SQLite, this technique performed worse than the Original technique, with a higher mean time taken across all 32 schemas for both. In the worst cases, this increase was up to ∼3 minutes and ∼67 minutes for HyperSQL and SQLite, respectively. When using PostgreSQL, Full schemata was more efficient for 91% of the schemas, saving up to 66 seconds compared to the Original technique in the best case. In the case of 2 of the 3 remaining schemas where the Full schemata technique is slower, the difference is relatively small, however for the iTrust schema it takes ∼20 minutes longer, therefore is likely to be of practical significance. Therefore, this technique is of limited benefit, only improving upon the Original technique when using PostgreSQL, and may be detrimental to efficiency for some schemas.

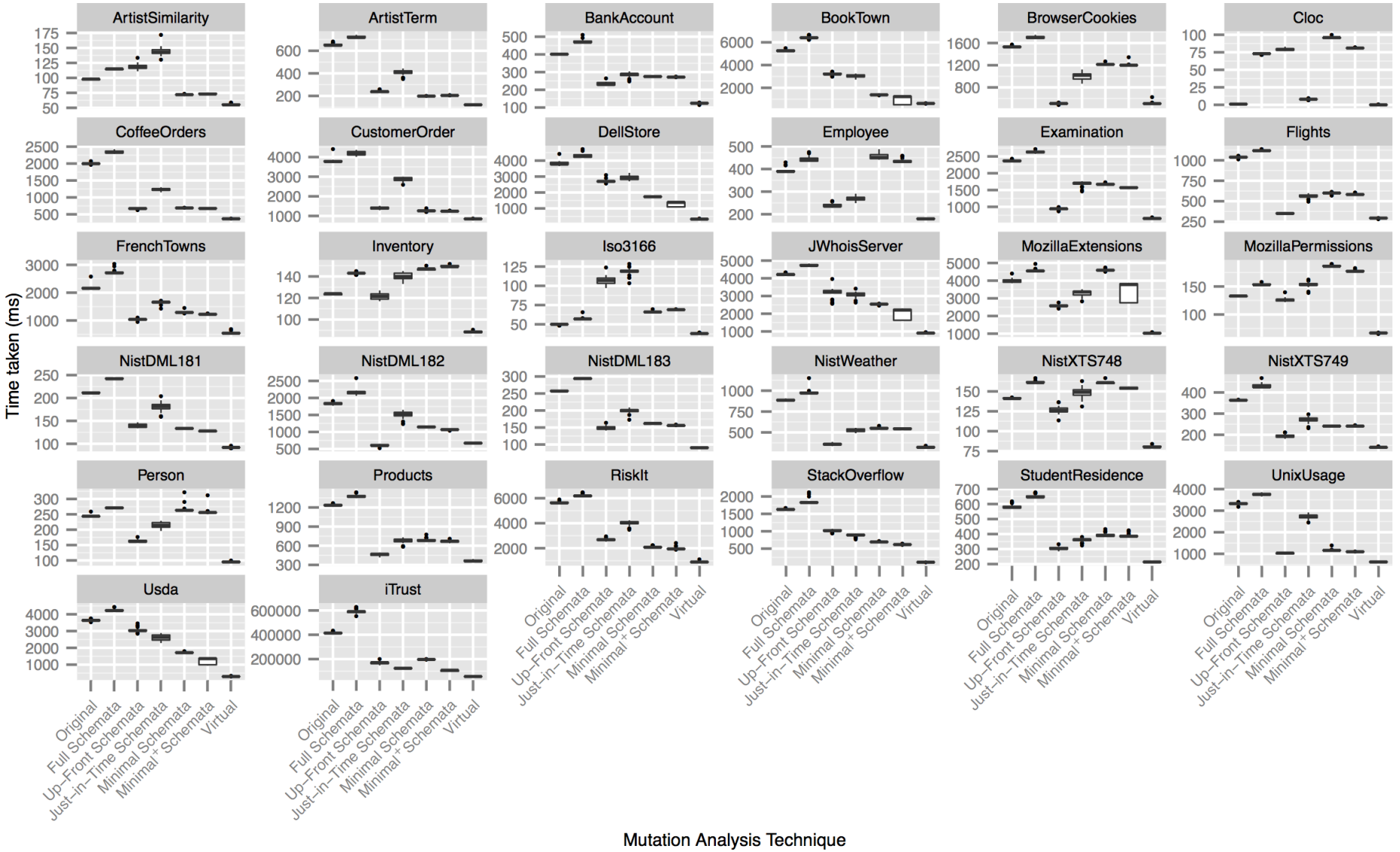


Figure 7.5: Time taken for mutation analysis for HyperSQL DBMS

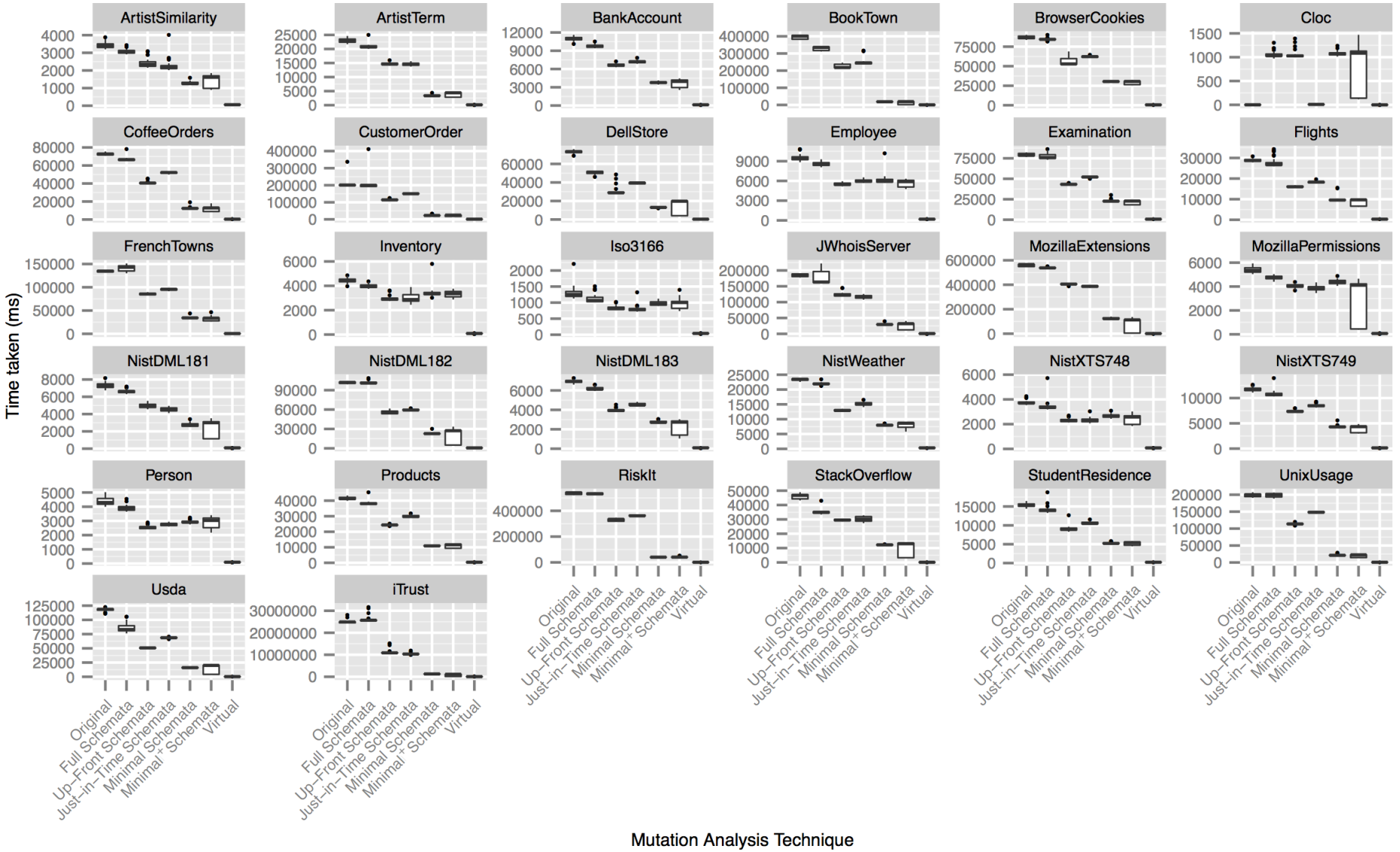


Figure 7.6: Time taken for mutation analysis for PostgreSQL DBMS.

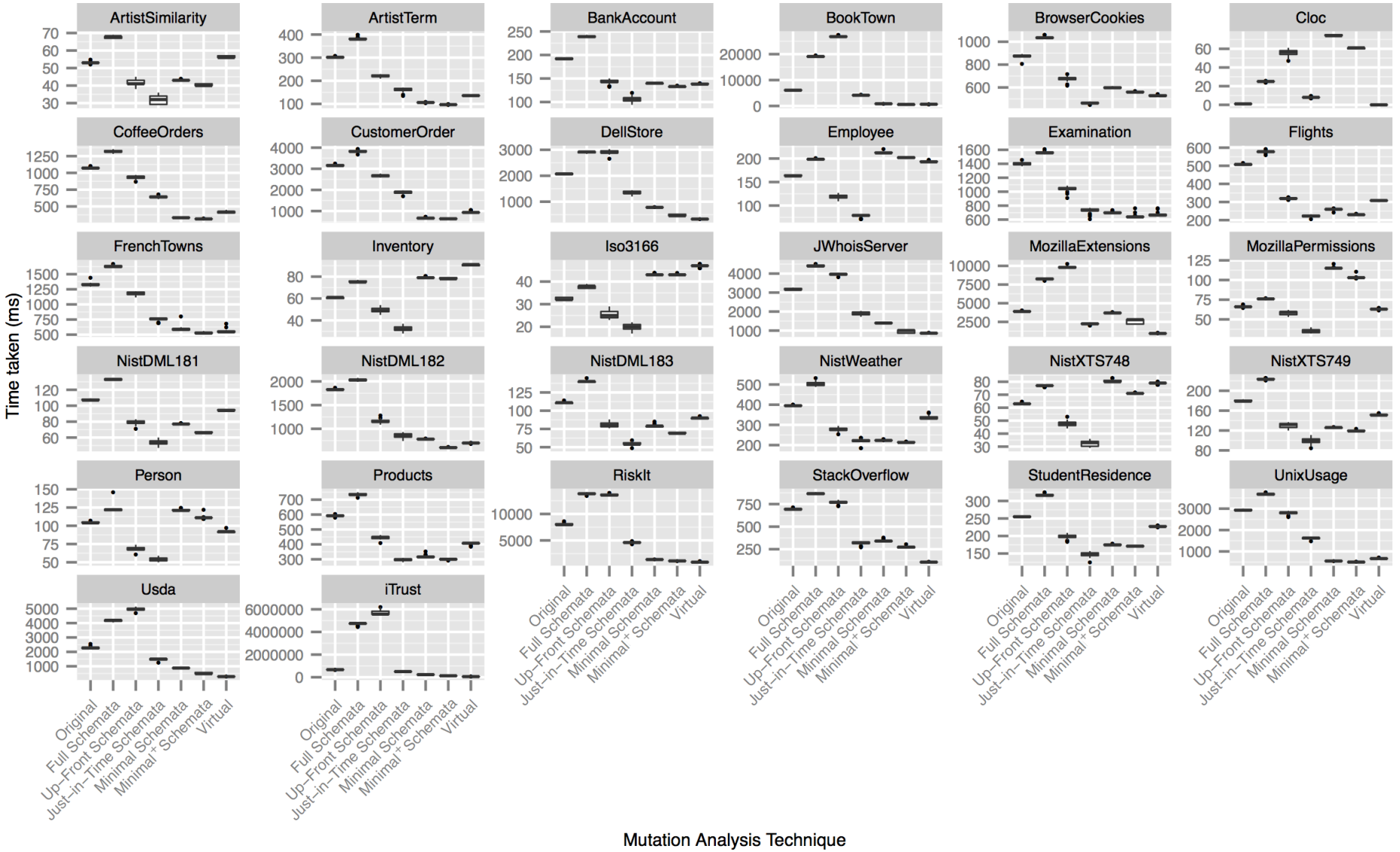


Figure 7.7: Time taken for mutation analysis for SQLite DBMS.

Table 7.2: Mean time taken (ms) for mutation analysis.

Highlighted times represent the lowest values in their respective rows.

DBMS	Schema	Technique						
		Original	Full schemata	Up-Front schemata	Just-in-Time schemata	Minimal schemata	Minimal$^+$ schemata	Virtual
HyperSQL	ArtistSimilarity	98	115	118	144	72	73	**55**
	ArtistTerm	653	722	238	409	198	204	**122**
	BankAccount	401	472	233	284	275	272	**124**
	BookTown	5263	6422	3219	3031	1382	1001	**628**
	BrowserCookies	1534	1703	**510**	998	1216	1205	512
	Cloc	1	73	79	8	96	81	**0**
	CoffeeOrders	1999	2340	670	1238	688	674	**369**
	CustomerOrder	3796	4177	1400	2868	1265	1242	**853**
	DellStore	3827	4303	2713	2915	1735	1291	**332**
	Employee	392	443	239	269	456	436	**180**
	Examination	2372	2632	941	1686	1674	1570	**656**
	Flights	1039	1120	351	561	600	584	**293**
	FrenchTowns	2173	2736	1034	1651	1295	1223	**553**
	Inventory	124	143	122	140	147	149	**89**
	Iso3166	50	57	107	119	66	69	**38**
	JWhoisServer	4220	4734	3167	3072	2541	2026	**906**
	MozillaExtensions	4000	4573	2575	3290	4598	3453	**1028**
	MozillaPermissions	133	153	126	153	187	177	**67**
	NistDML181	211	243	140	180	134	128	**92**
	NistDML182	1834	2165	**602**	1515	1148	1068	672
	NistDML183	257	294	149	199	162	156	**91**
	NistWeather	887	980	358	526	552	544	**322**
	NistXTS748	141	162	127	148	161	154	**80**
	NistXTS749	364	431	194	272	241	241	**141**
	Person	244	271	163	214	266	258	**95**
	Products	1237	1376	465	678	688	671	**364**
	RiskIt	5647	6214	2687	4002	2107	1976	**904**
	StackOverflow	1629	1849	1015	886	695	619	**110**
	StudentResidence	582	652	305	361	395	391	**214**
	UnixUsage	3323	3755	1027	2724	1169	1098	**625**
	Usda	3642	4241	3056	2618	1722	1238	**296**
	iTrust	415076	592731	168878	124094	196976	105451	**56369**
Postgres	ArtistSimilarity	3427	3082	2413	2283	1281	1460	**57**
	ArtistTerm	23055	20889	14740	14577	3409	3973	**122**
	BankAccount	10956	9782	6647	7239	3778	3704	**125**
	BookTown	396018	329902	224523	249077	18866	14059	**622**
	BrowserCookies	86805	84506	55734	62294	30386	29353	**509**
	Cloc	1	1060	1060	9	1078	845	**0**
	CoffeeOrders	72746	66864	40749	52050	12651	12175	**375**
	CustomerOrder	204817	204721	114883	150517	23410	22283	**862**
	DellStore	73100	50435	30503	39374	13034	14878	**332**
	Employee	9489	8611	5506	6002	6161	5684	**183**
	Examination	79400	77374	43327	52070	23063	21287	**661**
	Flights	28824	28025	16074	18297	9980	8693	**297**
	FrenchTowns	134387	141307	85588	95351	34519	32641	**543**
	Inventory	4444	3985	2958	3006	3442	3323	**89**
	Iso3166	1298	1125	831	809	977	970	**40**
	JWhoisServer	184691	178770	124084	116863	29746	26280	**896**
	MozillaExtensions	559688	538220	404378	386553	124270	80231	**1013**
	MozillaPermissions	5379	4744	4032	3846	4381	2942	**68**
	NistDML181	7318	6658	4944	4530	2769	2439	**91**
	NistDML182	101707	101514	55694	59513	22984	20160	**674**
	NistDML183	6938	6196	3975	4544	2752	2323	**89**
	NistWeather	23394	21967	12980	15151	7972	8006	**321**
	NistXTS748	3756	3456	2324	2326	2668	2428	**81**
	NistXTS749	11759	10836	7406	8530	4345	3992	**141**
	Person	4370	3911	2555	2759	2928	2879	**94**
	Products	41441	38375	24355	30034	10943	11012	**379**
	RiskIt	536439	532351	330288	362498	40099	41609	**881**
	StackOverflow	45975	35124	29535	30363	12256	9916	**109**
	StudentResidence	15346	14179	9121	10607	5260	5225	**213**
	UnixUsage	198718	198474	113680	148323	21912	19942	**636**
	Usda	117963	86286	50770	68638	16063	14973	**297**
	iTrust	24988581	26161311	11290218	10417891	1255578	801756	**56037**
SQLite	ArtistSimilarity	53	68	42	**32**	43	40	56
	ArtistTerm	302	382	221	161	106	**97**	136
	BankAccount	192	239	143	**106**	140	133	138
	BookTown	6103	19098	26792	4150	849	**545**	667
	BrowserCookies	874	1037	675	**463**	598	560	529
	Cloc	1	25	55	8	74	61	**0**
	CoffeeOrders	1074	1320	931	642	333	**314**	416
	CustomerOrder	3163	3827	2666	1882	668	**636**	946
	DellStore	2072	2912	2909	1348	782	477	**329**
	Employee	164	199	119	**78**	213	202	193
	Examination	1401	1561	1037	729	701	**646**	671
	Flights	508	578	320	**222**	259	230	308
	FrenchTowns	1331	1629	1179	754	597	**530**	556
	Inventory	61	75	49	**32**	79	78	91
	Iso3166	32	38	26	**20**	43	43	47
	JWhoisServer	3177	4408	3960	1897	1399	966	**864**
	MozillaExtensions	3900	8254	9819	2232	3705	2608	**955**
	MozillaPermissions	66	76	58	**35**	115	103	63
	NistDML181	108	133	79	**53**	77	66	94
	NistDML182	1830	2030	1160	851	783	**605**	701
	NistDML183	111	140	81	**55**	79	69	90
	NistWeather	396	503	277	221	224	**214**	337
	NistXTS748	63	77	48	**32**	81	71	79
	NistXTS749	180	224	130	**99**	126	120	152
	Person	105	123	68	**54**	122	111	92
	Products	593	734	445	**297**	318	300	406
	RiskIt	8045	13829	13612	4618	1390	1129	**901**
	StackOverflow	694	866	769	318	344	274	**107**
	StudentResidence	255	317	198	**147**	175	171	227
	UnixUsage	2933	3683	2798	1618	552	**503**	667
	Usda	2289	4179	4968	1476	877	512	**286**
	iTrust	669019	4708413	5716508	500727	228962	125281	**54412**

Table 7.3: Proportion of mean time taken for mutation analysis compared to Original technique.

Highlighted proportions represent the lowest values in their respective rows.

DBMS	Schema	Technique						
		Original	Full schemata	Up-Front schemata	Just-in-Time schemata	Minimal schemata	Minimal$^+$ schemata	Virtual
HyperSQL	ArtistSimilarity	1.000	1.173	1.204	1.469	0.735	0.745	**0.561**
	ArtistTerm	1.000	1.106	0.364	0.626	0.303	0.312	**0.187**
	BankAccount	1.000	1.177	0.581	0.708	0.686	0.678	**0.309**
	BookTown	1.000	1.220	0.612	0.576	0.263	0.190	**0.119**
	BrowserCookies	1.000	1.110	**0.332**	0.651	0.793	0.786	0.334
	Cloc	1.000	73.000	79.000	8.000	96.000	81.000	**0.000**
	CoffeeOrders	1.000	1.171	0.335	0.619	0.344	0.337	**0.185**
	CustomerOrder	1.000	1.100	0.369	0.756	0.333	0.327	**0.225**
	DellStore	1.000	1.124	0.709	0.762	0.453	0.337	**0.087**
	Employee	1.000	1.130	0.610	0.686	1.163	1.112	**0.459**
	Examination	1.000	1.110	0.397	0.711	0.706	0.662	**0.277**
	Flights	1.000	1.078	0.338	0.540	0.577	0.562	**0.282**
	FrenchTowns	1.000	1.259	0.476	0.760	0.596	0.563	**0.254**
	Inventory	1.000	1.153	0.984	1.129	1.185	1.202	**0.718**
	Iso3166	1.000	1.140	2.140	2.380	1.320	1.380	**0.760**
	JWhoisServer	1.000	1.122	0.750	0.728	0.602	0.480	**0.215**
	MozillaExtensions	1.000	1.143	0.644	0.822	1.149	0.863	**0.257**
	MozillaPermissions	1.000	1.150	0.947	1.150	1.406	1.331	**0.504**
	NistDML181	1.000	1.152	0.664	0.853	0.635	0.607	**0.436**
	NistDML182	1.000	1.180	**0.328**	0.826	0.626	0.582	0.366
	NistDML183	1.000	1.144	0.580	0.774	0.630	0.607	**0.354**
	NistWeather	1.000	1.105	0.404	0.593	0.622	0.613	**0.363**
	NistXTS748	1.000	1.149	0.901	1.050	1.142	1.092	**0.567**
	NistXTS749	1.000	1.184	0.533	0.747	0.662	0.662	**0.387**
	Person	1.000	1.111	0.668	0.877	1.090	1.057	**0.389**
	Products	1.000	1.112	0.376	0.548	0.556	0.542	**0.294**
	RiskIt	1.000	1.100	0.476	0.709	0.373	0.350	**0.160**
	StackOverflow	1.000	1.135	0.623	0.544	0.427	0.380	**0.068**
	StudentResidence	1.000	1.120	0.524	0.620	0.679	0.672	**0.368**
	UnixUsage	1.000	1.130	0.309	0.820	0.352	0.330	**0.188**
	Usda	1.000	1.164	0.839	0.719	0.473	0.340	**0.081**
	iTrust	1.000	1.428	0.407	0.299	0.475	0.254	**0.136**
Postgres	ArtistSimilarity	1.000	0.899	0.704	0.666	0.374	0.426	**0.017**
	ArtistTerm	1.000	0.906	0.639	0.632	0.148	0.172	**0.005**
	BankAccount	1.000	0.893	0.607	0.661	0.345	0.338	**0.011**
	BookTown	1.000	0.833	0.567	0.629	0.048	0.036	**0.002**
	BrowserCookies	1.000	0.974	0.642	0.718	0.350	0.338	**0.006**
	Cloc	1.000	1060.000	1060.000	9.000	1078.000	845.000	**0.000**
	CoffeeOrders	1.000	0.919	0.560	0.716	0.174	0.167	**0.005**
	CustomerOrder	1.000	1.000	0.561	0.735	0.114	0.109	**0.004**
	DellStore	1.000	0.690	0.417	0.539	0.178	0.204	**0.005**
	Employee	1.000	0.907	0.580	0.633	0.649	0.599	**0.019**
	Examination	1.000	0.974	0.546	0.656	0.290	0.268	**0.008**
	Flights	1.000	0.972	0.558	0.635	0.346	0.302	**0.010**
	FrenchTowns	1.000	1.051	0.637	0.710	0.257	0.243	**0.004**
	Inventory	1.000	0.897	0.666	0.676	0.775	0.748	**0.020**
	Iso3166	1.000	0.867	0.640	0.623	0.753	0.747	**0.031**
	JWhoisServer	1.000	0.968	0.672	0.633	0.161	0.142	**0.005**
	MozillaExtensions	1.000	0.962	0.723	0.691	0.222	0.143	**0.002**
	MozillaPermissions	1.000	0.882	0.750	0.715	0.814	0.547	**0.013**
	NistDML181	1.000	0.910	0.676	0.619	0.378	0.333	**0.012**
	NistDML182	1.000	0.998	0.548	0.585	0.226	0.198	**0.007**
	NistDML183	1.000	0.893	0.573	0.655	0.397	0.335	**0.013**
	NistWeather	1.000	0.939	0.555	0.648	0.341	0.342	**0.014**
	NistXTS748	1.000	0.920	0.619	0.619	0.710	0.646	**0.022**
	NistXTS749	1.000	0.922	0.630	0.725	0.370	0.339	**0.012**
	Person	1.000	0.895	0.585	0.631	0.670	0.659	**0.022**
	Products	1.000	0.926	0.588	0.725	0.264	0.266	**0.009**
	RiskIt	1.000	0.992	0.616	0.676	0.075	0.078	**0.002**
	StackOverflow	1.000	0.764	0.642	0.660	0.267	0.216	**0.002**
	StudentResidence	1.000	0.924	0.594	0.691	0.343	0.340	**0.014**
	UnixUsage	1.000	0.999	0.572	0.746	0.110	0.100	**0.003**
	Usda	1.000	0.731	0.430	0.582	0.136	0.127	**0.003**
	iTrust	1.000	1.047	0.452	0.417	0.050	0.032	**0.002**
SQLite	ArtistSimilarity	1.000	1.283	0.792	**0.604**	0.811	0.755	1.057
	ArtistTerm	1.000	1.265	0.732	0.533	0.351	**0.321**	0.450
	BankAccount	1.000	1.245	0.745	**0.552**	0.729	0.693	0.719
	BookTown	1.000	3.129	4.390	0.680	0.139	**0.089**	0.109
	BrowserCookies	1.000	1.186	0.772	**0.530**	0.684	0.641	0.605
	Cloc	1.000	25.000	55.000	8.000	74.000	61.000	**0.000**
	CoffeeOrders	1.000	1.229	0.867	0.598	0.310	**0.292**	0.387
	CustomerOrder	1.000	1.210	0.843	0.595	0.211	**0.201**	0.299
	DellStore	1.000	1.405	1.404	0.651	0.377	0.230	**0.159**
	Employee	1.000	1.213	0.726	**0.476**	1.299	1.232	1.177
	Examination	1.000	1.114	0.740	0.520	0.500	**0.461**	0.479
	Flights	1.000	1.138	0.630	**0.437**	0.510	0.453	0.606
	FrenchTowns	1.000	1.224	0.886	0.566	0.449	**0.398**	0.418
	Inventory	1.000	1.230	0.803	**0.525**	1.295	1.279	1.492
	Iso3166	1.000	1.188	0.812	**0.625**	1.344	1.344	1.469
	JWhoisServer	1.000	1.387	1.246	0.597	0.440	0.304	**0.272**
	MozillaExtensions	1.000	2.116	2.518	0.572	0.950	0.669	**0.245**
	MozillaPermissions	1.000	1.152	0.879	**0.530**	1.742	1.561	0.955
	NistDML181	1.000	1.231	0.731	**0.491**	0.713	0.611	0.870
	NistDML182	1.000	1.109	0.634	0.465	0.428	**0.331**	0.383
	NistDML183	1.000	1.261	0.730	**0.495**	0.712	0.622	0.811
	NistWeather	1.000	1.270	0.699	0.558	0.566	**0.540**	0.851
	NistXTS748	1.000	1.222	0.762	**0.508**	1.286	1.127	1.254
	NistXTS749	1.000	1.244	0.722	**0.550**	0.700	0.667	0.844
	Person	1.000	1.171	0.648	**0.514**	1.162	1.057	0.876
	Products	1.000	1.238	0.750	**0.501**	0.536	0.506	0.685
	RiskIt	1.000	1.719	1.692	0.574	0.173	0.140	**0.112**
	StackOverflow	1.000	1.248	1.108	0.458	0.496	0.395	**0.154**
	StudentResidence	1.000	1.243	0.776	**0.576**	0.686	0.671	0.890
	UnixUsage	1.000	1.256	0.954	0.552	0.188	**0.171**	0.227
	Usda	1.000	1.826	2.170	0.645	0.383	0.224	**0.125**
	iTrust	1.000	7.038	8.545	0.748	0.342	0.187	**0.081**

Table 7.4: The proportion of mean times taken for mutation analysis compared to Original technique, summarised as a mean across all schemas.

	Technique						
DBMS	Original	Full Schemata	Up-Front Schemata	Just-in-Time Schemata	Minimal Schemata	Minimal Minimal Schemata	Virtual
HyperSQL	1.000	1.151	0.627	0.808	0.689	0.644	**0.319**
Postgres	1.000	0.918	0.598	0.653	0.333	0.308	**0.010**
SQLite	1.000	1.542	1.313	**0.556**	0.662	0.586	0.615

Up-Front schemata: This technique was more efficient than the Original technique for the majority of schemas across all three DBMSs, reducing the average time taken for mutation analysis for 91%, 97% and 72% of schemas for HyperSQL, PostgreSQL and SQLite, respectively. For HyperSQL, in the best case Up-Front schemata saved 4.1 minutes, while in the worst case it increased the time taken by only 78ms, averaging to an overall time saving of approximately 1/3. This improvement was higher when using PostgreSQL, reducing the time taken by 3.8 hours in the best case, increasing it by 1 second in the worse case, and on average saving 40% of the time compared to the Original technique. However, for SQLite the Up-Front schemata technique only yielded a 670ms decrease in time taken in the best case, whilst taking 84 minutes longer in the worst case, which is 8.5 times higher the Original technique. As shown in Table 7.4, taking a mean across all schemas this equates to 30% slower than the Original technique. In summary, the Up-Front schemata technique provides a significant technique for most schemas when using either HyperSQL or PostgreSQL, but is not beneficial if using the SQLite DBMS.

Just-in-Time schemata: Regardless of which DBMS is used, this technique has at least similar performance to the Original technique, if not vastly better. The mean time taken for mutation analysis was faster for 97% of schemas for PostgreSQL and SQLite, and 81% for HyperSQL. In the worst cases, the mean time was 69ms, 8ms and 7ms higher than for the Original technique for HyperSQL, PostgreSQL and SQLite, respectively, while in the best cases ∼5 minutes, ∼4 hours and ∼3 minutes were saved, respectively. Over all DBMSs, the time saving was approximately 1/3, with a higher average time saving than any other technique when using SQLite. In summary, this technique consistently yields efficiency improvements over the Original technique, and is the most efficient choice when using SQLite.

Minimal schemata: When using the PostgreSQL DBMS, this technique reduces the

time taken for mutation analysis for all but one schema, Cloc, which was ∼1 second slower on average. However, this schema is an extreme example, where as there are no constraints the test data set generated is empty. As a result, those techniques which do prior transformation of the schema regardless of this (i.e., Up-Front schemata, Minimal schemata and Minimal$^+$ schemata) are likely to take longer than those techniques which do not, and can in essence return results with no work. In the best case when using PostgreSQL, the Minimal schemata technique reduces the time taken for mutation analysis by ∼6.6 hours, representing a saving of 95%, with an average reduction in time taken of ⅔. Although this technique only improves efficiency for 75% and 78% of schemas for HyperSQL and SQLite, respectively, those schemas where it does worse than the Original technique are generally small. As a result the prior transformation of the schema is a more significant proportion of the time spent, however, in real terms the additional time taken is generally small — 598ms and 73ms in the worst cases for HyperSQL and SQLite, respectively. In addition, the best cases reflect significant time savings of 3.6 minutes and 7.3 minutes, with overall averages over all schemas of 31% and 34%. Overall, the Minimal schemata technique provides a notable improvement in time taken for mutation analysis regardless of DBMS, especially for larger schemas.

Minimal$^+$ schemata: The results show that this technique outperforms the Minimal schemata technique for ∼88% of schemas across all DBMSs, which is expected given it attempts to improve upon Minimal schemata by reducing the number of constraints where possible. In addition, where the Minimal$^+$ schemata technique is less efficient the time taken is within ∼350ms of the Minimal schemata technique, and often less than 100ms different. Comparing to the Original technique, the Minimal$^+$ schemata technique is between 31% and 67% faster on average, depending on which DBMS is being used. In the best cases Minimal$^+$ schemata is 5.2 minutes, 6.7 hours and 9 minutes faster than the Original technique, while in the worst cases it is 80ms, 844ms and 60ms slower, on average, for HyperSQL, PostgreSQL and SQLite, respectively. In summary, the Minimal$^+$ schemata technique performs similarly or better than the Minimal schemata technique irrespective of the DBMS, with large potential time savings compared to the Original technique. Additionally, where slower than the Original technique the difference is often small, and likely to be practically unimportant.

Virtual: The Virtual mutation analysis technique was more efficient than the Original technique for 84% of schemas when using SQLite and 100% of schemas when using either

HyperSQL or PostgreSQL. However, of those 16% where the Virtual technique was slower, it took only an average of 30ms longer at most. As shown in Table 7.4, this technique achieved an average reduction in time taken of 68% for HyperSQL, 99% for PostgreSQL and 39% for SQLite. These correspond to average best case time savings of 6 minutes, 6.9 hours and 10 minutes, respectively. The results of Table 7.2 also show that the Virtual technique is more efficient than all other techniques for $^{30}/_{32}$ schemas for HyperSQL, $^{32}/_{32}$ schemas for PostgreSQL and $^{8}/_{32}$ schemas for SQLite. In the cases where the Virtual technique is slower than another technique, it is never by more than 310ms, which is a relatively small duration. Therefore, overall the Virtual mutation analysis technique is the most efficient, although does rely upon the correctness of the model of DBMS behaviour internal to SchemaAnalyst and thus cannot be used when introducing a new DBMS or additional DBMS functions, for example.

RQ2 Summary: *Of the novel techniques, only Full schemata does not yield efficiency improvements. The amount of time saved by each technique does depend on the DBMS being used. The Virtual technique is, on average, the most efficient for HyperSQL and PostgreSQL, while Just-in-Time schemata is the fastest technique for SQLite. However, the Virtual technique is close to matching this when averaging across all schemas, and therefore has the greatest efficiency.*

RQ3: *Is the Virtual mutation analysis technique more efficient than those which rely on communication with a DBMS?*

The results in Tables 7.2 and 7.3 show that for 94% and 100% of schemas for HyperSQL and PostgreSQL, respectively, the Virtual technique obtains the lowest mean time taken, whilst being only 2ms and 70ms slower than the fastest technique, Up-Front schemata, for the remaining 6% of schemas for HyperSQL. Table 7.4 reveals that on average the Virtual technique is ∼3.1 and ∼100 times faster than the Original technique for HyperSQL and PostgreSQL, respectively. In real terms this equates to ∼6 minutes and ∼7 hours faster, respectively, in the best case (when testing the iTrust schema).

However, when using the SQLite DBMS the Virtual mutation analysis technique is fastest for only 25% of schemas, compared to 47% for Just-in-Time schemata and 28% for Minimal^{+} schemata. Further examining the results of Table 7.2 with reference to Table 7.1 reveals that the complexity of the schema may influence which technique is the most efficient. For example, of the 5 schemas which had the highest mean time taken

for the Original technique, the Virtual technique was fastest for 4 while the Minimal$^+$ schemata was fastest for the other one. These schemas — iTrust, RiskIt, BookTown, MozillaExtensions and JWhoisServer — generally have high numbers of tables, columns and constraints, compared to other schemas, and therefore produce larger numbers of mutants. For these five schemas, the Virtual technique is at best ∼71 seconds faster than the next most efficient technique, Minimal$^+$ schemata, and ∼10 minutes quicker than the Original technique. For all schemas, where the Virtual technique is not fastest, it is at most 310ms slower (CustomerOrder) and only 65ms slower on average. This suggests that while the Virtual technique is not always the fastest, in practical usage it is likely the best choice as where it is better than other techniques the different is great, while where it is not the difference is relatively small.

RQ3 Summary: *In general, yes the Virtual technique is more efficient. If using SQLite, it may be marginally slower for small schemas, but is much faster for larger schemas.*

7.7 Summary

This Chapter described the algorithms and implementation of six different novel techniques for mutation analysis of relational database schemas, which apply the concepts of optimisations previously applied to mutation analysis of other artefacts to a new domain. These include the use of mutant schemata and parallelisation, as well as minimisation of the test artefact alongside a reduction of the number of tests executed. Each of these techniques was then compared to the existing, unoptimised Original technique, in terms of both accuracy (measured by the accuracy of the mutation score) and efficiency (according to the time taken). This involved an empirical experiment featuring 32 schemas, including many from real-world sources, and three DBMSs, to maximise the generalisability of the results. As all six novel techniques achieved the same mutation score as the Original technique across all schemas, DBMSs and repeat trials, only the relative efficiency was used to compare techniques. The Virtual technique generally performed similarly or better than the other five techniques, and consistently significantly reduced the time taken for mutation analysis compared to the Original technique. In the best case, this led to a time saving of almost 7 hours compared to the Original technique, when using PostgreSQL. Whilst both Just-in-Time schemata and Minimal$^+$ schemata

were faster than the Virtual technique when using SQLite, this difference was generally very small, and unlikely to be practically significant. However, these two techniques still remain valuable as a means of periodically checking the accuracy of the model of DBMS behaviour in SchemaAnalyst, to ensure the mutation score achieved by the Virtual technique is not made inaccurate when support is added for any additional DBMSs or SQL features.

Chapter 8

Conclusions and Future Work

8.1 Summary of Contributions

As outlined in Chapter 1, this thesis aimed to apply mutation analysis to evaluate test suites that were produced as a technique for schema testing, which attempt to insert data into a database as a means of exercising the integrity constraints of the SQL schema used to create it. This required a number of challenges to be addressed:

1. Determine how to produce schema mutants whose constraints contain possible programmer mistakes, including faults of omission and commission;
2. Investigate how the choice of DBMS may cause some mutants to be syntactically or semantically invalid – referred to as quasi-mutants;
3. Understand how the overlap of different SQL features may lead to the production of equivalent ($\texttt{mutant} \equiv \texttt{original}$) or redundant ($\texttt{mutant}_\texttt{x} \equiv \texttt{mutant}_\texttt{y}$) mutants;
4. Determine how to remove equivalent, redundant and quasi-mutants automatically, to prevent them impacting upon the effectiveness and efficiency of mutation analysis; and

5. The development of several techniques to reduce the computational cost of mutation analysis, by exploiting properties of database schemas and implementing domain-specific parallels of techniques used to optimise other applications of mutation analysis.

Chapter 3: "The SchemaAnalyst tool" described the design and implementation details of the SchemaAnalyst tool, which was used as a framework within which to implement this mutation analysis approach. Most importantly, this provided an intermediate representation of SQL that could be manipulated programmatically, which supported the specific SQL dialects of three different DBMSs. The mutation framework and mutation analysis portions of the SchemaAnalyst tool represent work that is entirely my own and therefore form a novel contribution of this thesis.

Chapter 4: "Mutation Operators for Relational Database Schemas" provided formal definitions for a collection of 14 mutation operators for injecting faults into the constraints of an SQL schema. These produced mutants by adding, removing or swapping columns in `PRIMARY KEY` (PKColumnA, PKColumnR, PKColumnE), `FOREIGN KEY` (FKColumnPairA, FKColumnPairR, FKColumnPairE) and `UNIQUE` constraints (UColumnA, UColumnR, UColumnE), adding and removing `NOT NULL` constraints (NNA and NNR), swapping relational operators and removing elements from `IN` lists in check expressions (CCRelopE and CInListElementR), and removing check constraints (CR). Implementing these operators in the SchemaAnalyst tool allowed a wide range of mutants to be generated that are able to represent schemas containing both faults of omission and commission for all major SQL constraints. In addition, for each of these operators expressions were given that can calculate how many mutants they will generate for a given schema mathematically based upon the schema's attributes, without having to execute the operator. This allows the number of mutants per operator to be determined ahead of mutation analysis, which may prove useful for the purposes of selective mutation (reducing the number of mutant operators applied) or mutant sampling (including only a certain number or proportion of mutants per operator).

Chapter 5: "Evaluating Coverage Criteria for Relational Database Schemas Using Mutation Analysis" makes use of the mutants generated by the prior Chapter to evaluate the data generation component otherwise implemented in the SchemaAnalyst tool. This compares the `INSERT` statements produced using two different data generation algorithms

– Random$^+$ and AVM– and according to nine coverage criteria, in terms of how many mutants they are able to kill. The latter consists of five constraint coverage criteria – APC, ICC, AICC, CondAICC and ClauseAICC – and four column coverage criteria – UCC, AUCC, NCC and ANCC. In addition, the ability of the data generated by each criteria to kill mutants produced by different operators is compared. A range of combinations of criteria are also compared according to mutation score, to determine how test suites can be combined to kill a higher proportion of mutants. Through the investigation of four research questions, this experiment revealed a number of useful results. Firstly, that AVM algorithm is consistently better at generating data to satisfy the coverage criteria than the Random$^+$ algorithm, especially for those criteria higher in the subsumption hierarchy – and therefore are generally more difficult to satisfy. Next, the discussion of the second research question showed that the coverage criteria that kills the most mutants depends upon the data generation algorithm applied – with ClauseAICC and AUCC killing the highest proportion of mutants for AVM and Random$^+$, respectively. Thirdly, that the constraint coverage criteria (APC, ICC, AICC, CondAICC, ClauseAICC) are better at killing mutants modelling faults of commission (where constraints are over-specified), while column coverage criteria (UCC, AUCC, NCC, ANCC) kill a higher proportion of those modelling faults of omission (where constraints are missing or under-specified). Finally, the fourth research question investigated whether coverage criteria could be combined from the constraint coverage criteria, null column coverage and unique column coverage categories, to give an overall increase in mutation score. Overall, the combination of ClauseAICC, UCC and ANCC was best if using the AVM algorithm, or ClauseAICC, AUCC and ANCC with the Random$^+$ algorithm, with a best case mean proportion of mutants killed of 94% compared to the best of 60% for a single criterion. These results suggest that to produce data that detects the highest proportion of faults the AVM algorithm and the combined set of coverage criteria is likely to be the best choice.

Chapter 6: "Automatically Identifying Ineffective Mutants" more closely explored the types of mutants produced by the schema integrity constraint operators, identifying three types of mutants that were detrimental to mutation analysis collectively described as ineffective mutants. Firstly, equivalent mutants are functionally identical to the original schema (`mutant` $\equiv$ `original`) and are a commonly studied occurrence when applying mutation analysis in other contexts. As there cannot exist a test case that returns a different result for the original and such a mutant, it cannot be killed during mutation analysis

and therefore artificially lowers the mutation score obtained from the truthful value, potentially concealing when a test suite is mutation adequate. These mutants therefore reduce the accuracy of mutation analysis, as well as increasing both the execution cost – as all tests must be executed for every equivalent mutant – and the human-cost associated with attempting to produce an adequate test suite. Secondly, redundant mutants are functionally identical to one or more other schema mutants ($\texttt{mutant}_x \equiv \texttt{mutant}_y$). These mutants may bias the possible mutation scores achieved by a test suite towards particular types of mutants, making it more difficult to compare two or more data generation techniques. In addition, computation time is wasted analysing each subsequent redundant mutant as by definition they are guaranteed to return the same results as each other. Finally, quasi-mutants are cases where the fault modelled by a mutation makes the schema syntactically or semantically invalid for at least one DBMS, but valid for another. This term is therefore intended to capture how a mutant may be still-born in some cases and not others, depending on the DBMS used. These mutants do not impact directly on mutation analysis, however their presence does preclude the application of some optimisations (discussed in a later Chapter). After defining a series of patterns that describe how to identify each of these types of ineffective mutant, applying DBMS-specific rules but without communicating with a DBMS, Chapter 6 evaluated how implementations of these rules could be applied after the generation of mutants to remove them entirely automatically. An empirical evaluation of this implementation showed that the static analysis approach used to detect quasi-mutants was significantly faster than executing the `CREATE TABLE` statements for a mutant with a DBMS, consistently taking multiple orders of magnitude less time. Removal of equivalent mutants was shown to often be time-cost effective, taking less time than the time saved by not analysing the mutants, meaning this improved both the efficiency and effectiveness of mutation analysis. Identifying and removing redundant mutants proved more expensive – as each mutant must be compared with each other – and therefore often increasing the time taken, as the infrequency of redundant mutants meant that time was rarely saved by removing them from the pool of mutants. However, it is arguable that paying this cost, which is on the scale of seconds, is worthwhile to improve the ability of mutation analysis to compare different data generation techniques with less bias towards certain mutants. In addition, if the prevalence of redundant mutants were to increase – for example when applying higher-level mutation – this may become time-cost effective.

Chapter 7: "Improving the Efficiency of Mutation Analysis for Relational Database

Schemas" investigated how different techniques for mutation analysis inspired by the literature (discussed in Section 2.3.5) can be applied when performing mutation of relational database schema integrity constraints. This led to the implementation of five algorithms that apply different optimisations to the problem, in an attempt to reduce the computational cost of execution. The *Full schemata* technique combines all mutant schemas into a single meta-mutant, reducing the number of `CREATE TABLE` statements executed from $|mutants|$ to 1, such that a specific mutant schema can be 'activated' by prefixing `INSERT` statements with a mutant identifier (e.g., '`m1_`'). The *Minimal schemata* and *Minimal*$^+$ *schemata* techniques then attempt to improve upon Full schemata by reducing the duplication of tables between mutants, such that only the mutated part of each is included. This requires the remapping of `FOREIGN KEY` constraints to a shared set of reference tables, whose state must be managed to ensure the correctness of results. Two parallel techniques were also developed, which make use of multiple connections to a DBMSs where possible (PostgreSQL) or multiple database instances (HyperSQL and SQLite). The *Just-in-Time schemata* algorithm creates the tables in the database for each mutant just prior to its analysis and removes them just after analysis, with the mutant tables prefixed with an identifier to avoid namespace collisions. Meanwhile, the *Up-Front schemata* technique creates a single database containing all mutants, much like the Full schemata approach. In both cases, the execution of the `INSERT` statements in the test suite is done in parallel, using a fixed number of threads in a pool. In addition to the five techniques inspired by existing optimisation strategies, a *Virtual mutation analysis* approach was developed. This removes the need for communicating with a DBMS, which can be expensive even when the database is held entirely in memory, by exploiting the fitness functions of the data generation component of SchemaAnalyst to model the expected behaviour of a target DBMS. An empirical evaluation using 32 schemas and all three supported DBMSs showed that all six of the techniques were able to obtain the same mutation scores as an unoptimised implementation. The efficiency of each technique depended on the DBMS chosen, with Virtual mutation being most efficient for the vast majority of schemas when using HyperSQL or PostgreSQL, while Just-in-Time schemata proved marginally faster when using SQLite.

8.2 Future Work

The work included in this thesis has investigated a number of problems faced when using mutation analysis to evaluate schema testing techniques, which test the integrity constraints of relational database schemas, and explored a variety of approaches to tackle these. This leads to a number of interesting opportunities for further work, the most significant of which are described below.

8.2.1 Schema mutants as models of real-world faults

As discussed in Section 2.3.1, mutation operators are used to inject syntactic faults into a software artefact to produce mutants that model possible programmer mistakes. As these are then used to evaluate the fault-finding capability of a test suite using mutation analysis, it is important that they accurately model the types of faults the programmer may make. While the 14 operators formally described and discussed in Chapter 4 are based upon both anecdotal evidence and personal intuitions about the types of mistakes made in integrity constraints, their accuracy in modelling real-world faults has not been empirically evaluated.

To determine the types of mistakes commonly made in integrity constraints, it may be possible to examine the changes made to constraints between different versions of a database schema in the version control history of an open source database application, such as in the approach of Curino et al. [28]. By tracking *schema evolution* in this way it may be possible to determine where constraints have been added or removed during development, which may represent cases where programmer faults were being corrected. Although it may prove difficult to identify these cases, such repositories may feature bug or issue trackers that contain extra information to help with this problem.

Once a likely case has been identified, the mutation operators could be applied to the version of the schema prior to the fix being applied to produce a set of mutants. Those mutants could then be compared to the fixed version of the schema using the equivalence detection code already implemented in Section 6.3.1. Where faults are produced from multiple changes, which could be detected automatically, this may involve generating higher-order mutants by applying multiple mutation operators (see Section 2.3.5). This

may lead to particular pairs or sequences of operators being identified that model a certain type of fault, which could be used in place of producing all higher-order mutants, which is generally prohibitively expensive. The higher the proportion of likely faults in the repository that can have corresponding mutants generated in this way, the better the mutation operators are at generating realistic faults.

8.2.2 Mutation and testing of column data types

While integrity constraints allow the schema designer to express a variety of limitations on the data that can be accepted into the database, even in a schema with no integrity constraints the data type specified for each column restricts the acceptable data. Referred to as *domain constraints* [100], these represent the most basic type of constraint applied to a column, and are mandatory in a SQL database schema. However, as with other types of constraint it is possible that a mistake by the schema designer leads to an error in an application using it to create a database. It therefore stands to reason that mutation operators that alter the type of a column may be useful to include when producing mutants to evaluate a given schema testing technique.

Given the large number of SQL types and the potentially huge number of mutants that could be generated when the number of columns in a schema increases, it may be necessary to investigate whether particular types of data type definition mistakes are made when designing a schema. This may require techniques to extract schema evolution information, as in the mentioned work of Curino et al. [28], to determine a smaller subset of data types that are commonly interchanged during development of a database application. Possible types with a higher likelihood of faults occurring may be where DBMSs handle certain types differently or between data types that are very similar, and therefore have subtly different use cases.

Where the mutants produced have seemingly incompatible types – for example, storing a decimal value in an integer column – the DBMS behaviour may make it difficult to kill them, as types may be coerced to the correct types prior to being stored. It would therefore likely be necessary to produce a coverage criterion that requires a data generation technique to create data of different types to then include in `INSERT` statements, in order to kill a high proportion of these mutants. Such a coverage criteria could then be

combined with the existing constraint, unique column and null column criteria to give a data generation configuration that can detect a wider range of faults in database schemas.

8.2.3 Database application schema mutation

The schema testing approach considered in this thesis assumes that the schema is being tested in isolation from other parts of an application. As such, the mutation operator have only been applied to a standalone schema. However, a database created using a schema is of little use without an application to interact with it, using it to store and retrieve data. Therefore, it may be useful to consider how mutation of a schema can be incorporated into the testing of a schema and application together.

Producing mutants of a schema embedded within an application may provide additional insight into the fault-finding capability of the application test suite, such as how effective data validation prior to accessing the database is at ensuring data being stored is valid according to the business logic of the application. This would help the tester gain additional confidence in the correctness of the behaviour of their application, by reducing the likelihood of undetected bugs being present.

8.2.4 Realistic test data generation

Once data is generated according to the criteria tested in Chapter 5, the schema testing approach we previously proposed [62] requires the tester to examine it to determine whether the sets of data accepted and rejected by the DBMS, according to the integrity constraints, match their intentions. Where they identify data wrongly categorised they must modify the constraints, adding to or removing from them as necessary to resolve the problem. This process should be repeated until the tester is convinced the DBMS returns the expected result for all items of data. However, as the data produced by the current data generation process in the SchemaAnalyst tool is initially seeded by random values or a set of default values, it is likely any data generated does not have certain characteristics of real data such as using string values that are easy for a human tester to read and recognise. Considering the tester must manually reason about each item of test data such readability is important as a means of reducing the human-time cost of

this testing approach. It may also be easier for the tester to consider data that closely models real data intended to be used in the application, as they may already be familiar with reasoning about this data.

Where data already exists for a particular schema – for example, if testing a schema that has previously been used as part of another application or for where a number of hand-derived test cases have been produced – this could be used to produce a pool of values that could be used to seed the data generation process. These values could then be modified as needed to meet the specific test requirements produced by the coverage criterion in use, according to a test data generation approach that changes the current proposed data such as the AVM algorithm. If data is not available for the schema under test, data types such as character strings could be seeded using dictionary values, which may still be easier for the tester to recognise quickly when manually inspecting the generated test cases. It may also be possible to include a measure of string readability as part of the fitness function used during data generation, which measures how close produced data is to meeting the test requirements given by a coverage criteria. Alternatively, this could be incorporated as a separate step after generating data values that satisfy the test requirements, post-processing the data in an attempt to improve its readability without altering its fitness according to the original fitness function.

Evaluating the effectiveness of these attempts to produce more realistic test data would likely require a human study, in which the time taken for human analysis of test data is measured with and without readable and realistic test data being used. This could be incorporated as part of a wider investigation into schema testing, in which individuals are given a description of the data for which they should design a schema and are provided with the SchemaAnalyst tool to assist them in identifying possible mistakes. The schemas created during this experiment would also be useful in determining where common mistakes are made, by recording the revisions the individual makes between different versions of their schema and asking them to identify the test cases they found useful in motivating these changes.

8.2.5 Scalability of mutation analysis for very large schemas

While the largest schema included in the empirical experiment of Chapter 7 is relatively complex – with 42 tables, 309 columns and 134 integrity constraints – it is still possible

that in a practical setting schema testing may be applied to larger schemas. As such, it is important to know how well the mutation analysis techniques presented in this thesis – particularly those techniques known to be the most efficient (Just-in-Time schemata, Minimal$^+$ schemata and Virtual mutation) – scale when the complexity of the schema increases further.

Similar scalability analysis has already been applied to the data generation component of SchemaAnalyst by Kinneer et al. [68]. In this approach, the complexity of the schema under test was increased by synthetically 'doubling' parts of it, such as the number of constraints present, and measuring the corresponding change in execution time, to determine whether it can be described by a linear function or one of a higher order. This doubling approach could be re-purposed for testing the scalability of mutation analysis in a similar manner, artificially scaling a schema and then measuring the time taken by each of the mutation analysis techniques previously identified as the most efficient.

As the size of the schema increases so should the generated test suite, to provide adequate coverage according to the coverage criteria within SchemaAnalyst. However, executing such a large number of test cases for each mutant is likely to become very expensive, thus making mutation testing infeasible. To mitigate this issue, it may become necessary to employ techniques such as test case prioritization [32] or test suite reduction [61], to reorder or remove tests such that execution cost is lowered. A conceptually perpendicular option may be to reduce the number of mutants produced for such schemas by employing selective mutation techniques (see Section 2.3.5), using an empirical analysis to determine a reduced set of operators that yield a mutation score that accurately models the score obtained by the full set of operators.

8.3 Concluding Remarks

The reliable storage of data forms an important part of a huge number of different applications, with relational databases being commonly employed to provide this functionality. Although prior research has focussed on testing both the DBMSs managing databases and the applications using them to store data the integrity constraints expressed in a schema, which provide the 'last line of defence' against the storage of incomplete or low

quality data, are not commonly tested. To evaluate such a schema testing technique, this thesis explored the use of mutation analysis to model faults in the constraints of a relational database schema. A set of mutation operators were proposed that modified each of the main types of constraint in an SQL schema. These were then used to compare a range of different data generation approaches for schema testing. Next, techniques were described to remove three types of ineffective mutant, which otherwise negatively impact upon both the effectiveness and efficiency of mutation analysis. Finally, a series of optimised mutation analysis algorithms were described, implemented and evaluated to improve its efficiency. Overall, the work of this thesis therefore provides an automated approach for evaluating test suites produced by schema testing techniques that is both effective for comparing different approaches and computationally efficient, such that its use in both a practical setting and large scale academic experiments is feasible.

Chapter 9

Bibliography

[1] SQL as understood by SQLite. `http://www.sqlite.org/lang_createtable.html#primkeyconst`. [Online; accessed 24-June-2014].

[2] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, Atlanta GA, September 1979.

[3] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation*, volume 3103 of *Lecture Notes in Computer Science*. 2004.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[6] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608 –624, aug. 2006.

[7] ANSI/ISO/IEC International Standard – ISO/IEC 9075-2:1999. Database Language SQL — Part 2: Foundation (SQL/Foundation), 1999.

[8] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 685–696, New York, NY, USA, 2011. ACM.

[9] Arvind Arasu, Raghav Kaushik, and Jian Li. DataSynth: Generating Synthetic Data using Declarative Constraints. *Proceedings of the 37th International Conference Very Large Data Bases*, 4(12):1418–1421, 2011.

[10] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[11] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[12] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval*, 2011.

[13] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 506–515. IEEE, 2007.

[14] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 341–352, New York, NY, USA, 2007. ACM.

[15] Carsten Binnig, Donald Kossmann, and Eric Lo. Multi-RQP: Generating Test Databases for the Functional Testing of OLTP Applications. In *Proceedings of the 1st International Workshop on Testing Database Systems*, DBTest '08, pages 5:1–5:6, New York, NY, USA, 2008. ACM.

[16] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[17] Mattias Brybo. A mutation testing tool for java programs, 2003. Masters Thesis, Department of Numerical Analysis and Computer Science, KTH Royal Institute Of Technology.

[18] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[19] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–233, 1980.

[20] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.

[21] Brandon Butler. Amazon: Our cloud powered Obama's campaign. *Network World*, 2012.

[22] Choi Byoungju and Aditya P Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135 – 152, 1993.

[23] Mark Anthony Cachia, Mark Micallef, and Christian Colombo. Addressing practical challenges of mutation testing. In *Validation Strategies for Software Evolution (VSSE)*, 2013.

[24] W. K. Chan, S. C. Cheung, and T. H. Tse. Fault-based testing of database application programs with conceptual data model. In *Proceedings of the 5th International Conference on Quality Software*, 2005.

[25] John J. Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5), 1994.

[26] Jake Cobb, Gregory M. Kapfhammer, James A. Jones, and Mary Jean Harrold. Dynamic invariant detection for relational databases. In *Proceedings of the 9th International Workshop on Dynamic Analysis*, 2011.

[27] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13, 1970.

[28] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *International Conference on Enterprise Information Systems (ICEIS)*, 2008.

[29] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th International Workshop on the Automation of Software Test*, 2010.

[30] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), 1978.

[31] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. Compiler-integrated program mutation. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference, (COMPSAC)*, pages 351–356, sep 1991.

[32] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), 2006.

[33] M. Ellims, D. Ince, and Marian Petre. The Csaw C mutation tool: Initial results. In *Testing: Academic and Industrial Conference Practice and Research Techniques (Mutation 2007 workshop)*, pages 185–192, Sept 2007.

[34] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[35] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[36] Gordon Fraser and Franz Wotawa. Mutant minimization for model-checker based test-case generation. In *Testing: Academic and Industrial Conference Practice and Research Techniques (Mutation 2007 workshop)*, pages 161–168. IEEE, 2007.

[37] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292, 2012.

[38] B.J.M. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *4th International Workshop on Mutation Analysis*, 2009.

[39] B.P. Gupta, D. Vira, and S. Sudarshan. X-Data: Generating test data for killing sql mutants. In *Proceedings of the 26th International Conference on Data Engineering*, 2010.

[40] Szymon Guz. Basic mistakes in database testing. `http://java.dzone.com/articles/basic-mistakes-database`. (Accessed 24/01/2014).

[41] Florian Haftmann, Donald Kossmann, and Eric Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1), 2007.

[42] M. Harman, Yue Jia, and W.B. Langdon. A manifesto for higher order mutation testing. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 80–89, April 2010.

[43] Mark Harman, Rob Hierons, and Sebastian Danicic. Mutation testing for the new century. chapter The Relationship Between Program Dependence and Mutation Analysis, pages 5–13. 2001.

[44] Mark Harman, Yue Jia, and William B Langdon. Strong higher order mutation-based test data generation. In *Foundations of software engineering (FSE)*, pages 212–222. ACM, 2011.

[45] Rob Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[46] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[47] Sean A Irvine, Tin Pavlinic, Leonard Trigg, John G Cleary, Stuart Inglis, and Mark Utting. Jumble Java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques (Mutation 2007 workshop)*, pages 169–175, 2007.

[48] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, April 2002.

[49] J.-M. Jazequel and B. Meyer. Design by contract: the lessons of ariane. *Computer*, 30(1):129–130, 1997.

[50] O.A. Jefferson, A.J. Offutt, and R.J. Untch. Mutation 2000: Uniting the orthogonal. *Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, 2001.

[51] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[52] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011.

[53] Yue Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, Sept 2008.

[54] Yue Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic and Industrial Conference Practice and Research Techniques (Mutation 2008 workshop)*, pages 94–98, Aug 2008.

[55] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST 2011, pages 50–56, New York, NY, USA, 2011. ACM.

[56] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 50–56, November 2011.

[57] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the 7th Workshop on Mutation Analysis*, 2012.

[58] René Just, Michael D Ernst, and Gordon Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. *arXiv preprint arXiv:1303.2784*, 2013.

[59] René Just, Michael D Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 315–326. ACM, 2014.

[60] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering*, 2014.

[61] G.M. Kapfhammer. Towards a method for reducing the test suites of database applications. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012.

[62] Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2013.

[63] Kalpesh Kapoor. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering*, 2(2):80–87, 2006.

[64] S.A. Khalek, B. Elkarablieh, Y.O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[65] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 25th International Conference on Automated Software Engineering*, 2010.

[66] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, June 1991.

[67] Cody Kinneer, Gregory M. Kapfhammer, Chris J. Wright, and Phil McMinn. Automatically evaluating the efficiency of search-based test data generation for relational database schemas. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015.

[68] Cody Kinneer, Gregory M. Kapfhammer, Chris J. Wright, and Phil McMinn. expOse: Inferring worst-case time complexity by automatic empirical study. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015.

[69] W.B. Langdon, M. Harman, and Yue Jia. Multi objective higher order mutation testing with genetic programming. In *Testing: Academic and Industrial Conference Practice and Research Techniques (Mutation 2009 workshop)*, pages 21 –29, 2009.

[70] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416 – 2430, 2010.

[71] Richard J. Lipton and Frederick Gerald Sayward. The status of research on program mutation. pages 355–373, December 1978.

[72] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[73] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: A mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[74] L. Madeyski. Judy – a mutation testing tool for Java. *IET Software*, 4:32–42, February 2010. ISSN 1751-8806.

[75] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, 1991.

[76] Phil McMinn, Chris J. Wright, and Gregory M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology*, 25(1):8:1–8:49, 2015.

[77] Phil McMinn, Gregory M. Kapfhammer, and Chris J. Wright. Virtual mutation analysis of relational database schemas. In *International Workshop on the Automation of Software Test*, 2016. (Accepted).

[78] Andrew Meneely, Ben Smith, and Laurie Williams. Appendix b: iTrust electronic health care system case study. *Software and Systems Traceability*, page 425, 2012.

[79] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing Verification and Reliability*, 9(4):205–232, 1999.

[80] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[81] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[82] A. Offutt. The coupling effect: Fact or fiction. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, pages 131–140, 1989.

[83] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[84] A. Jefferson Offutt. A practical system for mutation testing: Help for the common programmer. In *Proceedings of the 1994 International Conference on Test*, pages 824–830, 1994.

[85] A. Jefferson Offutt and W. Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[86] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[87] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[88] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.

[89] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for java. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, September 2004.

[90] Thomas J. Ostrand and Elaine J. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289 – 300, 1984.

[91] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011.

[92] Kai Pan, Xintao Wu, and Tao Xie. Automatic test generation for mutation testing on database applications. In *8th International Workshop on Automation of Software Test*, pages 111–117, May 2013.

[93] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. *37th International Conference on Software Engineering (ICSE)*, 2015.

[94] Meikel Poess and John M. Stephens, Jr. Generating Thousand Benchmark Queries in Seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, volume 30 of *VLDB '04*, pages 1045–1053. VLDB Endowment, 2004.

[95] PostgreSQL Project. About PostgreSQL. `http://www.postgresql.org/about/`. (Accessed 17/12/2012).

[96] D. Schuler and A. Zeller. (un-)covering equivalent mutants. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[97] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of the 7th Joint Meeting on Foundations of Software Engineering*, 2009.

[98] 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 *Proceedings of the 27th International Conference on Data Engineering*, 2011.

[99] M. L. Shooman and M. I. Bolsky. Types, distribution, and test and correction times for programming errors. In *Proceedings of the International Conference on Reliable Software*, pages 347–357, 1975.

[100] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 6th edition, 2010.

[101] Donald R. Slutz. Massive stochastic testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998.

[102] SQLite Developers. Most widely deployed SQL database engine. `http://www.sqlite.org/mostdeployed.html`, . (Accessed 21/11/2012).

[103] SQLite Developers. The SQLite DBMS. `http://www.sqlite.org/`, . (Accessed 1/7/2014).

[104] John M. Stephens and Meikel Poess. Mudd: a multi-dimensional data generator. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, pages 104–109, 2004.

[105] The hsql Development Group. HSQLDB homepage. `http://hsqldb.org/`. (Accessed 23/06/2015).

[106] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. SQLMutation: A tool to generate mutants of SQL database queries. In *Proceedings of the Second Workshop on Mutation Analysis*, 2006.

[107] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. Mutating database queries. *Information and Software Technology*, 49(4), 2006.

[108] Javier Tuya, M. José Suárez-Cabal, and Claudio de la Riva. Query-Aware Shrinking Test Databases. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 6:1–6:6, 2009.

[109] UK Transparency and Open Data team. Data.gov.uk. `http://data.gov.uk/`. (Accessed 31/07/2015).

[110] Roland H. Untch. Mutation-based software testing using program schemata. In *Proceedings of the 30th annual Southeast regional Conference*, ACM-SE 30, pages 285–291, New York, NY, USA, 1992. ACM.

[111] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT International symposium on Software Testing and Analysis*, ISSTA '93, pages 139–148, New York, NY, USA, 1993. ACM.

[112] U.S. General Services Administration. Data.gov. `https://www.data.gov/`. (Accessed 31/07/2015).

[113] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[114] K.S. How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2–3):119 – 161, 2003.

[115] Richard Y. Wang and Diane M. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of Management Information Systems*, 12(4): 5–33, 1996.

[116] W. Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31:185 – 196, 1995.

[117] Chris J. Wright, Gregory M. Kapfhammer, and Phil McMinn. Efficient mutation analysis of relational database structure using mutant schemata and parallelisation. In *Proceedings of the 8th International Workshop on Mutation Analysis*, 2013.

[118] Chris J Wright, Gregory M Kapfhammer, and Phil McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Proceedings of the 14th International Conference on Quality Software*, 2014.

[119] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th*

International Conference on Software Engineering (ICSE), pages 919–930. ACM, 2014.

[120] Chixiang Zhou and Phyllis Frankl. JDAMA: Java database application mutation analyser. *Software Testing, Verification and Reliability*, 21(3), 2011.

Appendix A

Using the Mutation Framework

A.1 Introduction

This appendix describes how the mutation framework, discussed in Section 3.6, can be used to perform experiments such as those in the contribution Chapters of this thesis. These experiments make use of the mutation operators of Chapter 4, optionally the ineffective mutant removal techniques detailed in Chapter 6, and either the Original mutation technique described in Section 1.3.2 or an optimised mutation analysis approach from Chapter 7. Both the mutation framework and the SchemaAnalyst itself will be made available online via the SchemaAnalyst website (`http://schemaanalyst.org/`). The appendix is divided into three Sections:

Section A.2: "Shared Configuration" briefly details the global configuration files in SchemaAnalyst, which are specify values for options used in multiple places across the tool, including the mutation framework.

Section A.3: "Generating Mutants" describes how mutants can be generated without performing mutation analysis, such that the number produced under different configurations (e.g., with and without ineffective mutants) can be compared.

Section A.4: "Executing Mutation Analysis" details how to execute the mutation analysis component of the framework to evaluate the quality of a test suite,

including using the data generation component to produce SQL `INSERT` statements and describing how to make use of the optimisations discussed in the prior Chapters of this thesis.

A.2 Shared Configuration

The global configuration files for SchemaAnalyst, residing in `config/`, are Java properties files that are used as a singular point to specify various options for parts of the tool, such that they do not need to be passed as arguments to all relevant commands. To allow users to maintain their own copies of these files that are not committed to the version control system, the configuration system automatically loads versions suffixed with `.local` if they exist (e.g., `database.properties.local` takes precedence over `database.properties`). There are three main configuration files used by SchemaAnalyst– `database`, `locations` and `logging`. The `database` file contains options relating to DBMS communication, `locations` is used to determine where various files important to SchemaAnalyst are located, and `logging` contains options relating to the `java.util` logging system. The properties contained in each are described in the tables below:

Properties file	database
Property	**Description**
dbms	The DBMS to use (note: this may affect parts of the tool that do not connect to a DBMS but that must have DBMS-specific behaviour, such as ineffective mutant removal).
postgres_port	The port used to connect to Postgres.
postgres_username	The username used to connect to Postgres.
postgres_password	The password used to connect to Postgres.
postgres_host	The network host used to connect to Postgres.
postgres_database	The name of the Postgres database.
postgres_driver	The class name for the Postgres JDBC driver.
sqlite_path	The directory to store the SQLite database files in.
sqlite_driver	The class name for the SQLite JDBC driver.
sqlite_in_memory	Whether to use the faster 'in-memory' mode of SQLite.
hsqldb_path	The directory to store the HyperSQL database files in.
hsqldb_username	The username used to connect to HyperSQL.
hsqldb_password	The password used to connect to HyperSQL.
hsqldb_driver	The class name for the HyperSQL JDBC driver.
hsqldb_in_memory	Whether to use the faster 'in-memory' mode of HyperSQL.

Properties file	locations
Property	**Description**
lib_dir	Where the required libraries (.jar) files are located.
src_dir	Where the Java source files are located.
build_dir	Where the compiled Java class files are located.
dist_dir	Where the distributable .jar file should be saved.
dist_name	The name of the .jar distributable file.
config_dir	Where the configuration files are located.
database_dir	Where database files should be stored.
results_dir	Where results files should be stored.
schema_src_dir	Where schema SQL files are located.
case_study_src_dir	Where the Java files produced by parsing schema files should be stored.
case_study_package	The package parsed schemas will be stored.
test_src_dir	Where the tests for SchemaAnalyst are located.
test_package	The name of the test package for SchemaAnalyst.

Properties file	logging
Property	**Description**
handlers	The java.util logging handlers to use.
.level	The logging level to use (e.g., WARNING, INFO, FINE).

A.3 Generating Mutants

To avoid the cost of running the entire mutation analysis process to measure the number of mutants produced by a particular set of operators, or how many are discarded by the ineffective mutant removal process, the mutation framework includes functionality to generate mutants without executing them. This is implemented in the `AnalysePipeline` class within the `mutation.analysis.util` package, which given the name of a mutation pipeline will create a report detailing the number of mutants produced for each operator, removed by each ineffective mutant removal phase and the overall number of effective mutants remaining for each operator. The command is executed using the following parameters:

Class	`org.schemaanalyst.mutation.analysis.util.AnalysePipeline`	
Parameter name	**Required**	**Description**
`casestudy`	✓	The class name of the schema to use, which has been parsed into the SchemaAnalyst intermediate representation.
`dbms`	✓	The DBMS to use (Postgres|SQLite|HyperSQL).
`mutationPipeline`		The mutation pipeline to use to generate mutants (default: `AllOperatorsWithRemovers`).
`outputfolder`		The directory to output results into (default: as specified in configuration file).

By default, this will produce a file at the location `results/analysepipeline.dat`, which is in a comma-separated value (CSV format) and usually contains many lines of results per execution. As an example, consider the following mock output:

results/analysepipeline.dat:

```
dbms,casestudy,pipeline,type,operator,count,timetaken
Postgres,parsedcasestudy.ArtistSimilarity,AllOperatorsWithRemovers,produced,NNCA,3,1
Postgres,parsedcasestudy.ArtistSimilarity,AllOperatorsWithRemovers,removed,EquivalentMutantRemover,1,27
Postgres,parsedcasestudy.ArtistSimilarity,AllOperatorsWithRemovers,retained,NNCA,2,NA
```

The first three columns specify the dbms, case study (or schema) and pipeline used, respectively – Postgres, ArtistSimilarity and AllOperatorsWithRemovers in this case. The type column describes what sort of data is listed in the given column, which must be `produced`, `removed` or `retained` (the number of mutants remaining after removal). Next, the class name of the mutation operator or remover applied is listed – in this case, the `NNCA` (`NOT NULL` column addition) was used to create three mutants, one of which was removed for being equivalent, leaving two mutants remaining. The final column lists the time taken in milliseconds, for either production or removal.

A.4 Executing Mutation Analysis

To create data to exercise the integrity constraints of a schema, using the SchemaAnalyst data generation component, and then perform mutation analysis to evaluate it, the `MutationAnalysis` class from the `mutation.analysis.executor` package is used. This is a highly configurable class whose runtime parameters can be used to set a number of options for both the data generation phase and the mutation analysis phase. The available parameters are detailed in the table below:

Class	`org.schemaanalyst.mutation.analysis.executor.MutationAnalysis`	
Parameter name	**Required**	**Description**
`casestudy`	✓	The class name of the schema to use, which has been parsed into the SchemaAnalyst intermediate representation.
`criterion`		The coverage criterion to use to generate data for.
`dataGenerator`		The data generator to use to produce SQL `INSERT` statements.
`maxevaluations`		The maximum fitness evaluations for the search algorithm to use.
`randomseed`		The seed used to produce random values for the data generation process.
`mutationPipeline`		The mutation pipeline to use to produce and, optionally, remove mutants.
`technique`		The mutation technique to use (e.g., original, fullSchemata, minimalSchemata).
`useTransactions`		Whether to use SQL transactions to improve the performance of a technique, if possible.

Executing this class produces a single results file in CSV format that contains one line per execution, located at `results/newmutationanalysis.dat`. This contains a number of fields:

dbms The DBMS.

casestudy The schema.

criterion The integrity constraint coverage criterion.

datagenerator The data generation algorithm.

randomseed The value used to seed the pseudo-random number generator.

coverage The level of coverage the produced data achieves according to the criterion.

evaluations The number of fitness evaluations used by the search algorithm.

tests The number of test cases in the produced test suite.

mutationpipeline The mutation pipeline used to generate mutants.

scorenumerator The number of mutants killed by the generated data.

scoredenominator The total number of mutants used for mutation analysis.

technique The mutation analysis technique.

transactions Whether SQL transactions were applied, if possible.

testgenerationtime The time taken to generate test data in milliseconds.

mutantgenerationtime The time taken to generate mutants in milliseconds.

originalresultstime The time taken to execute the test suite against the non-mutated schema.

mutationanalysistime The time taken to perform analysis of all of the mutant schemas.

timetaken The total time taken by the entire process.

Due to architectural differences between the Virtual mutation analysis technique and other techniques, it must instead be executed using the `MutationAnalysisVirtual` class within the same package, which produces results in the same format. In addition, if it is necessary to retrieve information about the execution time for each mutant individually, the `MutantTiming` technique is provided. This is a specialised version of the Original technique that is instrumented to record this additional information.

A.5 Summary

This appendix has described how to run two important parts of the mutation framework – mutant generation through the `AnalysePipeline` class, and mutation analysis through the `MutationAnalysis` and `MutationAnalysisVirtual` classes. These were used to produce the results of the empirical studies in the main Chapters of this thesis, and may also be used to perform other experiments related to the mutation analysis of relational database schemas.