Driected Random

Abdullah Alsharif

3 January 2017

Directed Random Mechanism

Directed Random works as same as random technique however it is guided by predicates that are checked then fixing a solution. For instance, each INSERT statement must comply to a test requirement that have one or many predicates such as NOT NULL for a specific column, first directed random generate random insert statement then proceed to fixing the insert statement based on a given predicate and if it does not comply to the predicate. If a column is NULL but the predicate require a NOT NULL for this specific column, directed random will fix the insert statement to have NOT NULL. However, Directed random usually fixes one predicate at a time, so if there is many violated predicates in one insert statement it will only fix one then iterate to the next evaluation to fix the other remaining predicates. This means that each evaluation does not fix all predicates or search for optimal solution, however each evaluation checks if the statement is complying with the test requirement.

Directed Random Algorithm

```
p <= predicate
n <= insert statement
CHECK method:
  IF n Comply with p THEN
    return true
  ELSE
    return false
END METHOD

FIX method:
    GET non-Complied predicate
    GET c <=column for non-complied predicate
    REPEAT:
      generate random value for column
    UNTIL vaule comply with predicate
END METHOD

generate random values for table insert n
result <= CHECK n aganist p

while result == Ture
  FIX n aganist p
  result <= CHECK n aganist p
END WHILE
```

In our experiments we ran two test data generators AVM and Directed Random (DR), from our experiment we are looking at the performance of the two techniques in regard of test generation timing and mutation score. This will help us to determine which of the techniques are better in those both factors. Looking at test generation timing will determine which of the two are faster in generating test cases. On the other hand we will look at mutation analysis of the two techniques to determine the strength and the capability of the test suite generated to detect faults.

Experiment Set Up

Our experiment set-up was to run each technique 30 times for each case study using one combined coverage criteria "ClauseAICC+AUCC+ANCC". Each run has different random seed to see the difference of results.

Case Studies

The following table has the case studies that are been used in our experiment:

Table 1: Table continues below

Schema	Tables	Columns	Total Columns	Total Constraints
ArtistSimilarity	2	3	3	3 (0)
ArtistTerm	5	7	7	7 (0)
BankAccount	2	9	9	8 (0)
BookTown	22	67	67	28 (1)
BrowserCookies	2	13	13	10 (4)
Cloc	2	10	10	0 (0)
CoffeeOrders	5	20	20	19 (0)
CustomerOrder	7	32	32	42 (1)
DellStore	8	52	52	39 (0)
Employee	1	7	7	4 (0)
Examination	2	21	21	9 (0)
Flights	2	13	13	10 (4)
FrenchTowns	3	14	14	24 (1)
Inventory	1	4	4	2 (0)
Iso3166	1	3	3	3 (0)
iTrust	42	309	309	134 (15)
JWhoisServer	6	49	49	50 (0)
MozillaExtensions	6	51	51	7 (4)
MozillaPermissions	1	8	8	1 (0)
NistDML181	2	7	7	2 (2)
NistDML182	2	32	32	2 (2)
NistDML183	2	6	6	2 (2)
NistWeather	2	9	9	13 (6)
NistXTS748	1	3	3	3 (0)
NistXTS749	2	7	7	7 (1)
Person	1	5	5	7 (1)
Products	3	9	9	14 (1)
RiskIt	13	57	57	36 (1)
StackOverflow	4	43	43	5 (0)
StudentResidence	2	6	6	8 (0)
UnixUsage	8	32	32	24 (1)
Usda	10	67	67	31 (0)
Total	172	975	975	554 (47)

Table 2: Table continues below

CHECK Constraints	FOREIGN KEY Constraints	NOT NULL Constraints
0 (0)	2 (0)	0
0 (0)	4 (0)	0
0 (0)	1 (0)	5

CHECK Constraints	FOREIGN KEY Constraints	NOT NULL Constraints
2 (1)	0 (0)	15
2 (1)	1 (1)	4
0 (0)	0 (0)	0
0 (0)	4 (0)	10
1 (1)	7 (0)	27
0 (0)	0 (0)	39
3 (0)	0 (0)	0
6 (0)	1 (0)	0
1 (1)	1 (1)	6
0 (0)	2 (0)	13
0 (0)	0 (0)	0
0 (0)	0 (0)	2
8 (8)	1 (0)	88
0 (0)	0 (0)	44
0 (0)	0 (0)	0
0 (0)	0 (0)	0
0 (0)	1 (1)	0
0 (0)	1 (1)	0
0 (0)	1 (1)	0
5 (5)	1 (0)	5
1 (0)	0 (0)	1
1 (0)	1 (0)	3
1 (1)	0 (0)	5
4 (0)	2 (0)	5
0 (0)	10 (0)	15
0 (0)	0 (0)	5
3 (0)	1 (0)	2
0 (0)	7 (0)	10
0 (0)	0 (0)	31
38 (18)	49 (5)	335

PRIMARY KEY Constraints	UNIQUE Constraints
1 (0)	0 (0)
3 (0)	0 (0)
2 (0)	0 (0)
11 (0)	0 (0)
2 (1)	1 (1)
0 (0)	0 (0)
5 (0)	0 (0)
7 (0)	0 (0)
0 (0)	0 (0)
1 (0)	0 (0)
2 (0)	0 (0)
2 (2)	0 (0)
0 (0)	9 (1)
1 (0)	1 (0)
1 (0)	0 (0)
37 (7)	0 (0)
6 (0)	0 (0)
2 (0)	5 (4)
1 (0)	0 (0)

PRIMARY KEY Constraints	UNIQUE Constraints
1 (1)	0 (0)
1 (1)	0 (0)
0 (0)	1 (1)
2 (1)	0 (0)
0 (0)	1 (0)
2 (1)	0 (0)
1 (0)	0 (0)
3 (1)	0 (0)
11 (1)	0 (0)
0 (0)	0 (0)
2 (0)	0 (0)
7 (1)	0 (0)
0 (0)	0 (0)
114 (17)	18 (7)

Results

Test generation Timing

When comparing test generation time we look at how efficient the technique are in regard of the time it takes to generate test suites (ALL AVM and DR has 100% coverage). Figure 1 shows the average test generation timing for each technique for each DBMS, for all runs and schemas. Just By looking at the graph it shows that Directed Random is much faster/efficient compared to AVM in generating test cases nearly 1 second faster for different SQL database engine. ??? Why Postgres takes longer for each AVM and DR compared to other engines ? different semantics or larger engine ?

To look in more details we split test generation timing analysis for each case study. In Figure 2 we review average test generation timing for each techinque for each schema split by DBMSs and for all runs. We can see that Directed Random still winning for each schema. By looking at all of the results in Figure 2 we can see that DR is better than all AVM even by fractions of seconds.

In Figure 3 I show the spread of values of test generation times in regard of DBMS and technique using a box plot, for all runs and schemas. In this plot we sum all result for each run and spread the values in the box plot, this will help to evaluate the spread of runs for all schema and for each technique split by database engine. As shown in the plot that DR is takes less time compared to AVM in generating test.

In Figure 4 I show the spread of values for test generation timing for each schema, DBMS and technique, for all runs. This will help us seeing the spread of values for each case study and how long it takes to generate test cases.

Mutation Scores

In Figure 5 I shows the average mutation score for each technique for each DBMS, for all runs and schemas. Just By looking at the graph it shows that Directed Random is batter when compared to AVM in killing more mutants.

In Figure 6 I review average mutation score for each techinque for each schema split by DBMSs, for all runs. We can see that Directed Random have a better or similar scores to AVM however not even one schema has less score comparing to AVM.

In Figure 7 I show the spread of values of mutation score in regard of DBMS and technique using a box plot, for all runs and schemas.

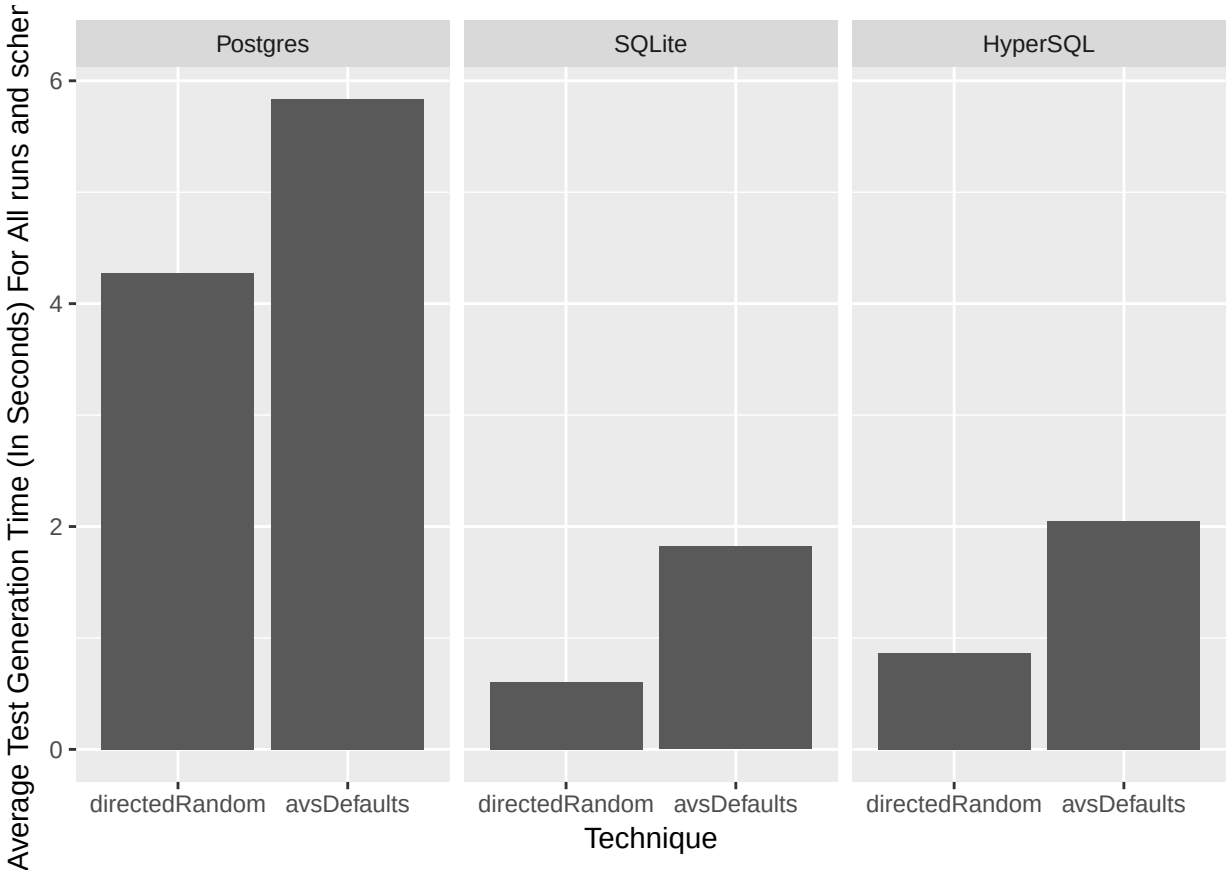

Figure 1: Avrages of Test generation timing - in seconds

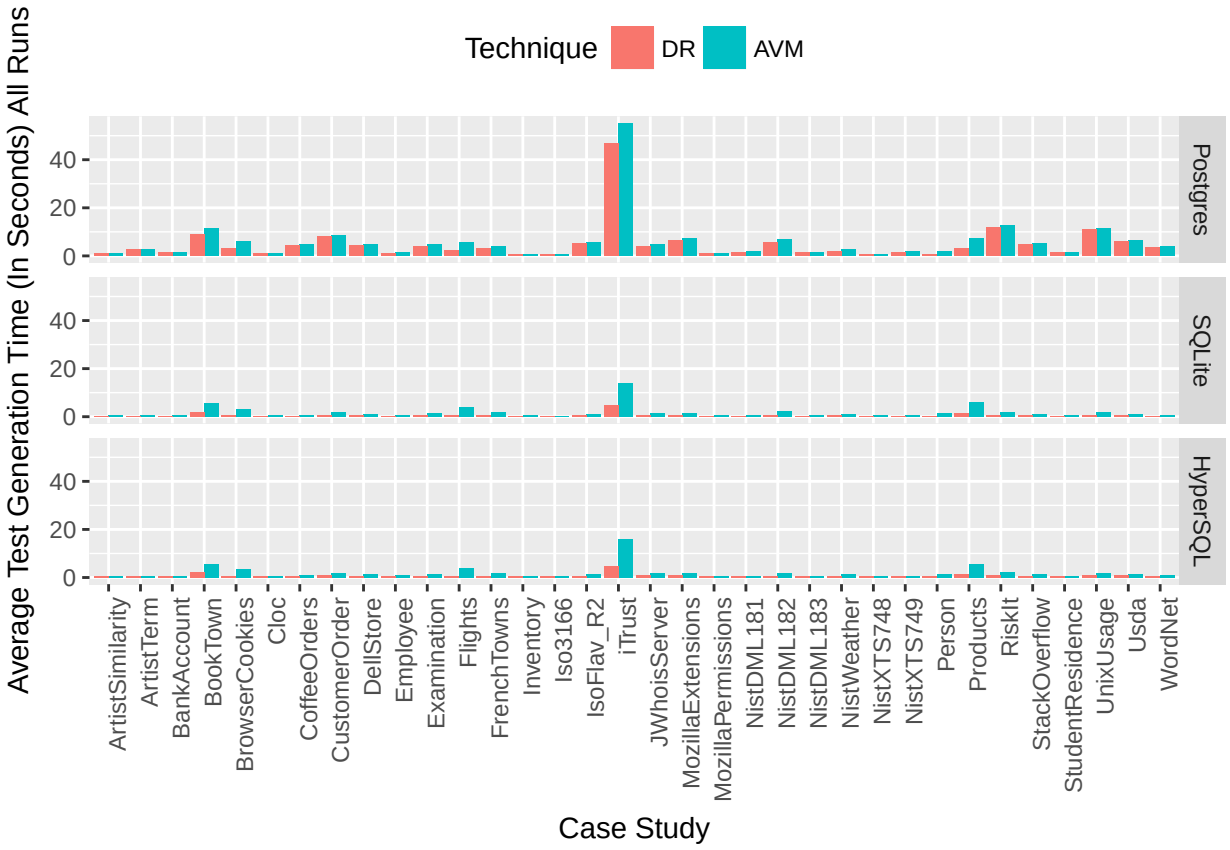

Figure 2: Avrages of Test generation timing for each schema- in seconds

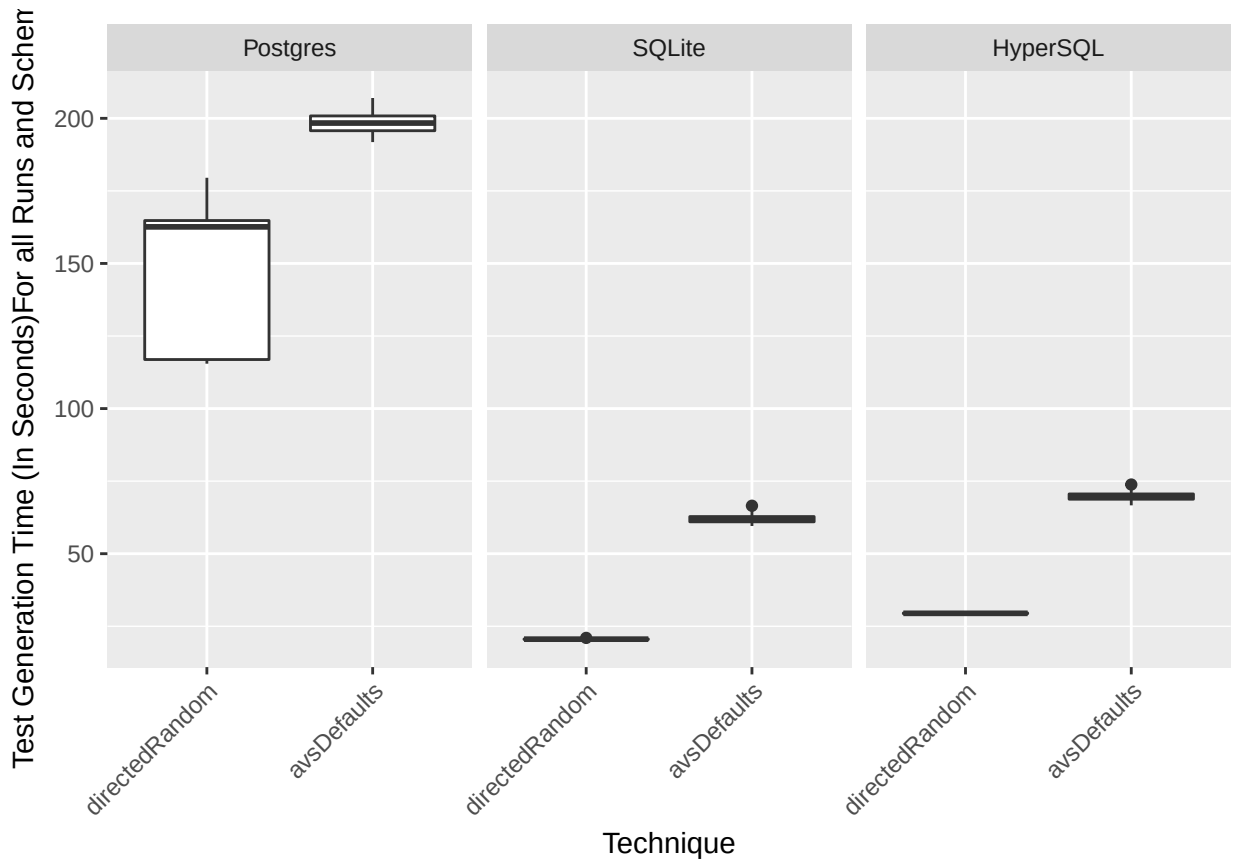

Figure 3: Test generation timing - in seconds

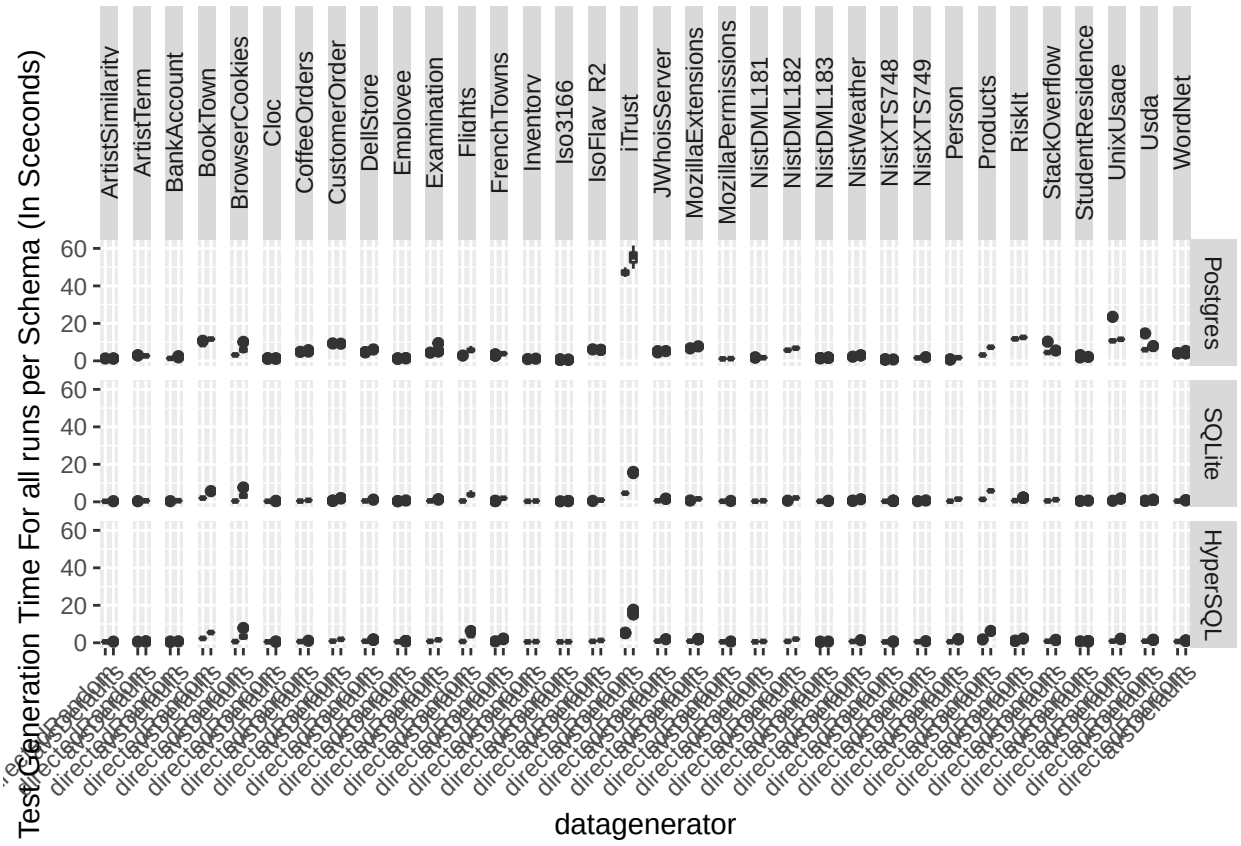

Figure 4: Box plot for mutation scores for DBMS, techniques and schemas - in precentage

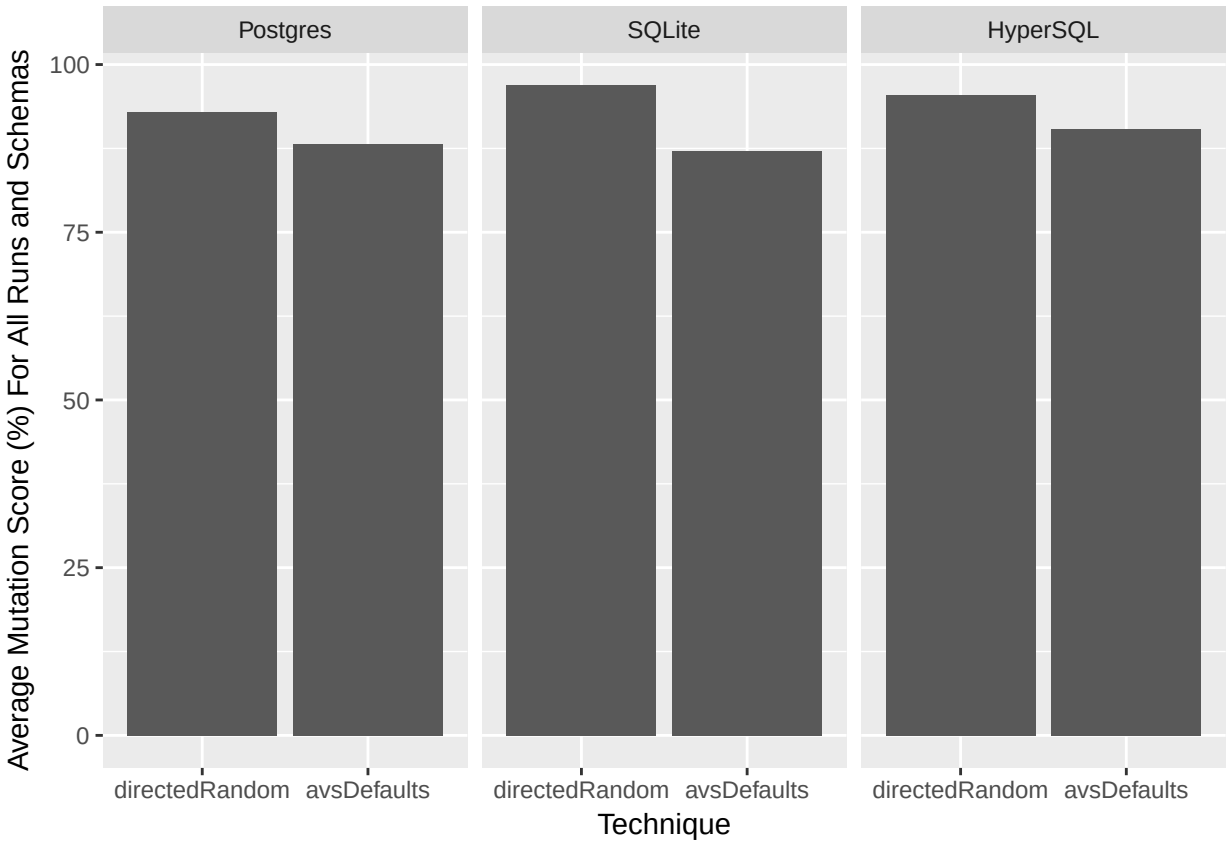

Figure 5: Avrages of Mutation Score - in precentage

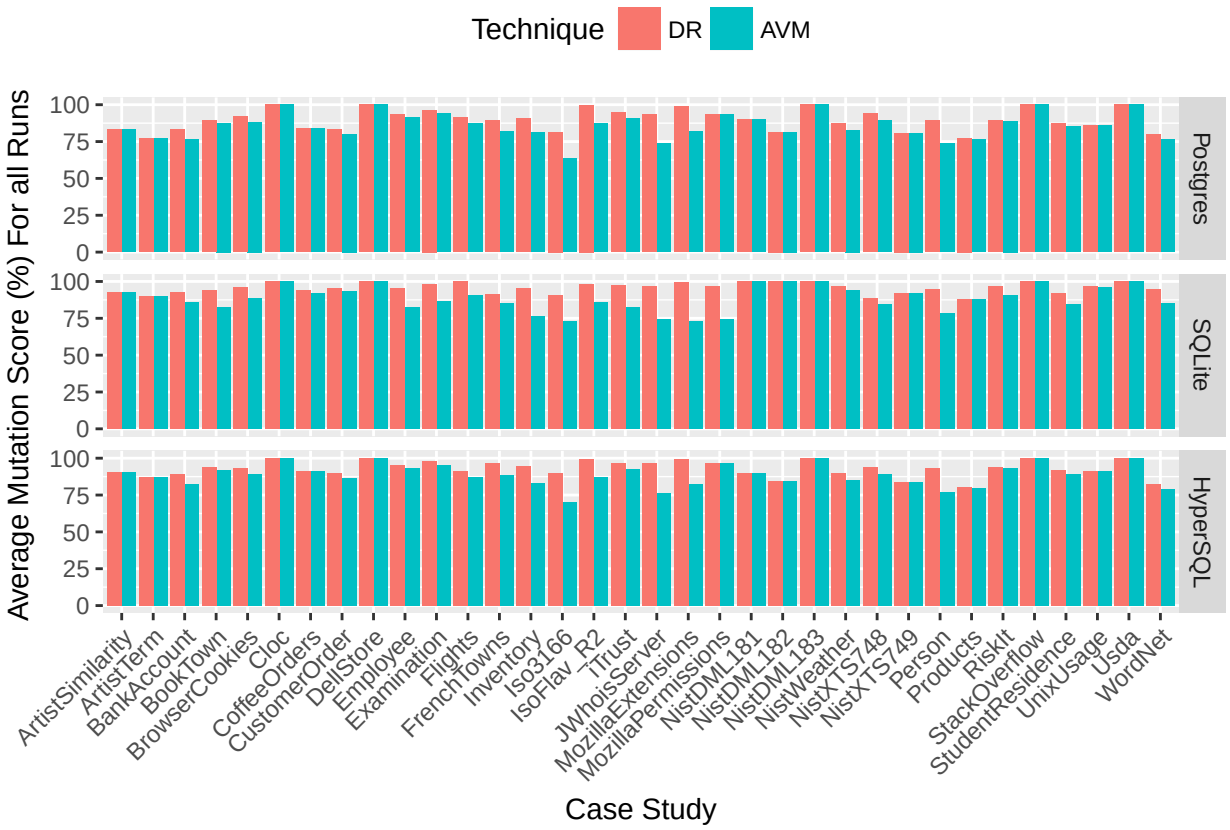

Figure 6: Avrages of Mutation Score for each schema - in precentage

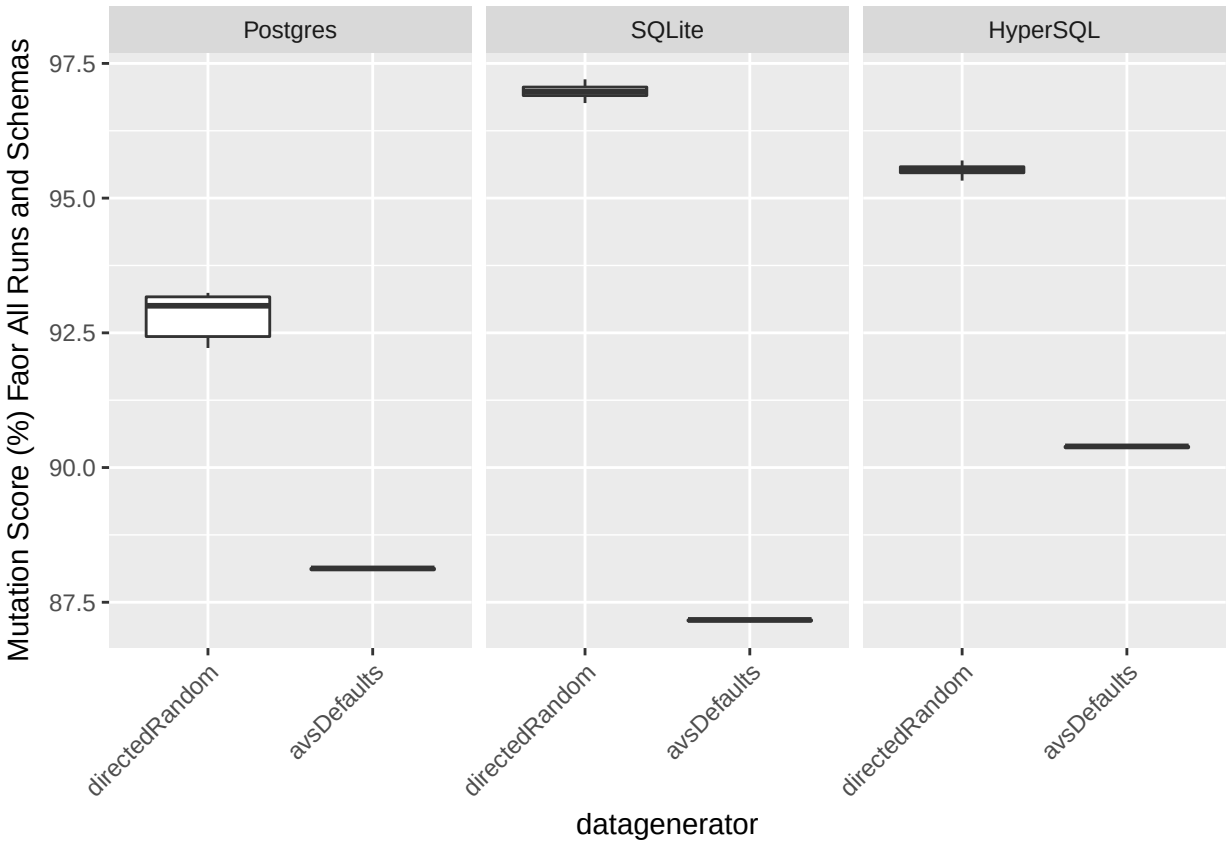

Figure 7: Box plot for mutation scores for DBMSs and techniques - in precentage

In Figure 8 I show the spread of values for mutation scores for each schema, DBMS and technique, for all runs.

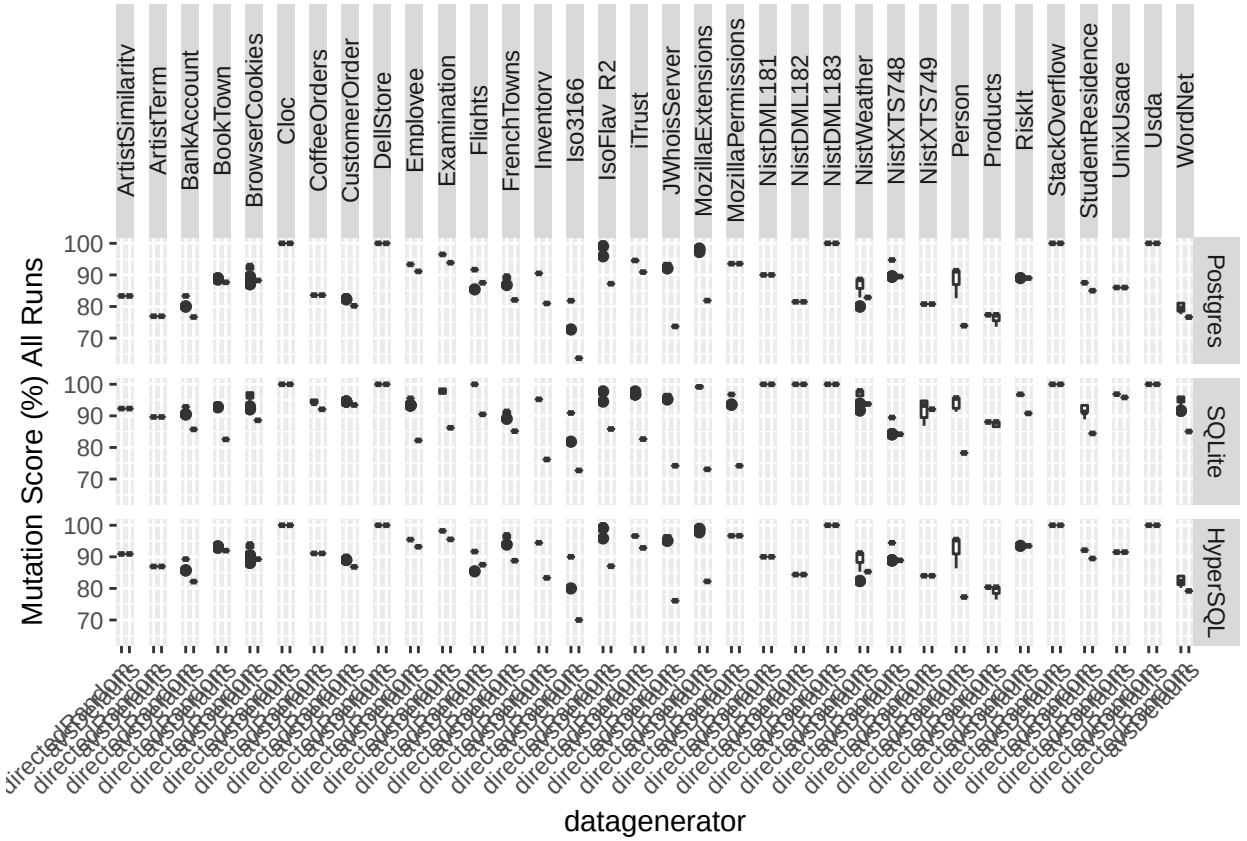

Figure 8: Box plot for mutation scores for DBMS, techniques and schemas - in precentage

Mutation Scores and Mutation Operators

In Figure 9 I shows the average mutation score for each technique for each DBMS and the mutatan operators, for all runs, schemas and not including Equvilant, Redundant Quasi mutants . Just By looking at the graph it shows that Directed Random is batter when compared to AVM in killing more mutants for two operators the rest shows same percentages.

In Figure 10 I shows a box plot mutation score for each technique for each DBMS and the mutatan operators, this will help us see the spread of values. Just By looking at the graph it shows that Directed Random is batter when compared to AVM in killing more mutants for two operators the rest shows same percentages.

HeatMap Plot of mutant analysis per technique for each Operator for all schemas

To analysis mutation score in more details we look at the mutant operators that are been killed by both techniques and for each DB engine. In Figure 11 We show that the average percentages of all schemas for each operator that is been killed for each technique. The figure shows that DR has always a higher percentage kill for all operators when compared to AVM in all DB engines which conclude that DR is much superior to AVM technique.

To look at why PKColumnA has a low kills for AVM, we investigated manually for the reason. We regenerated the test suites with same random seed, with first look we found that AVM has many empty Strings when

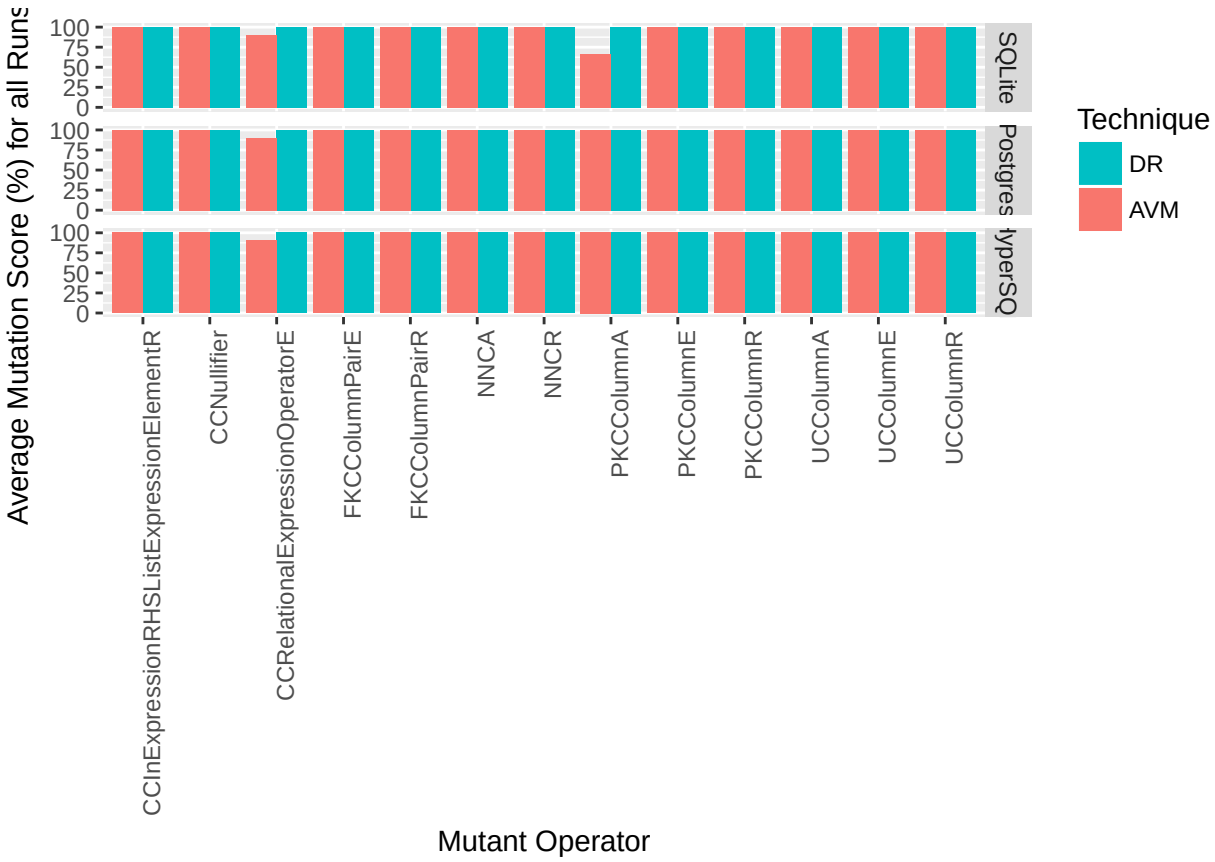

Figure 9: Averages of mutant scores in regard of Mutant Operators and DBMSs - in precentage

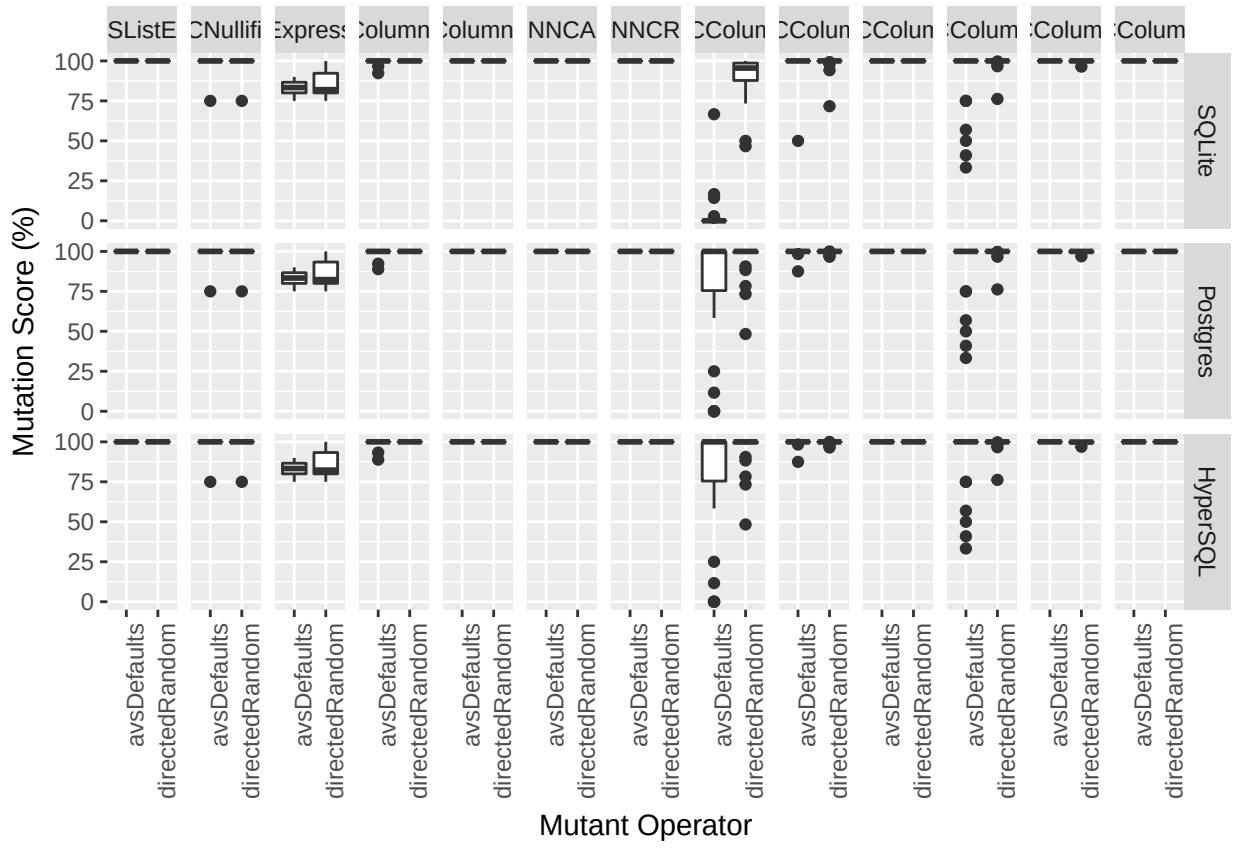

Figure 10: Box Plot of mutant scores in regard of Mutant Operators and DBMSs - in precentage

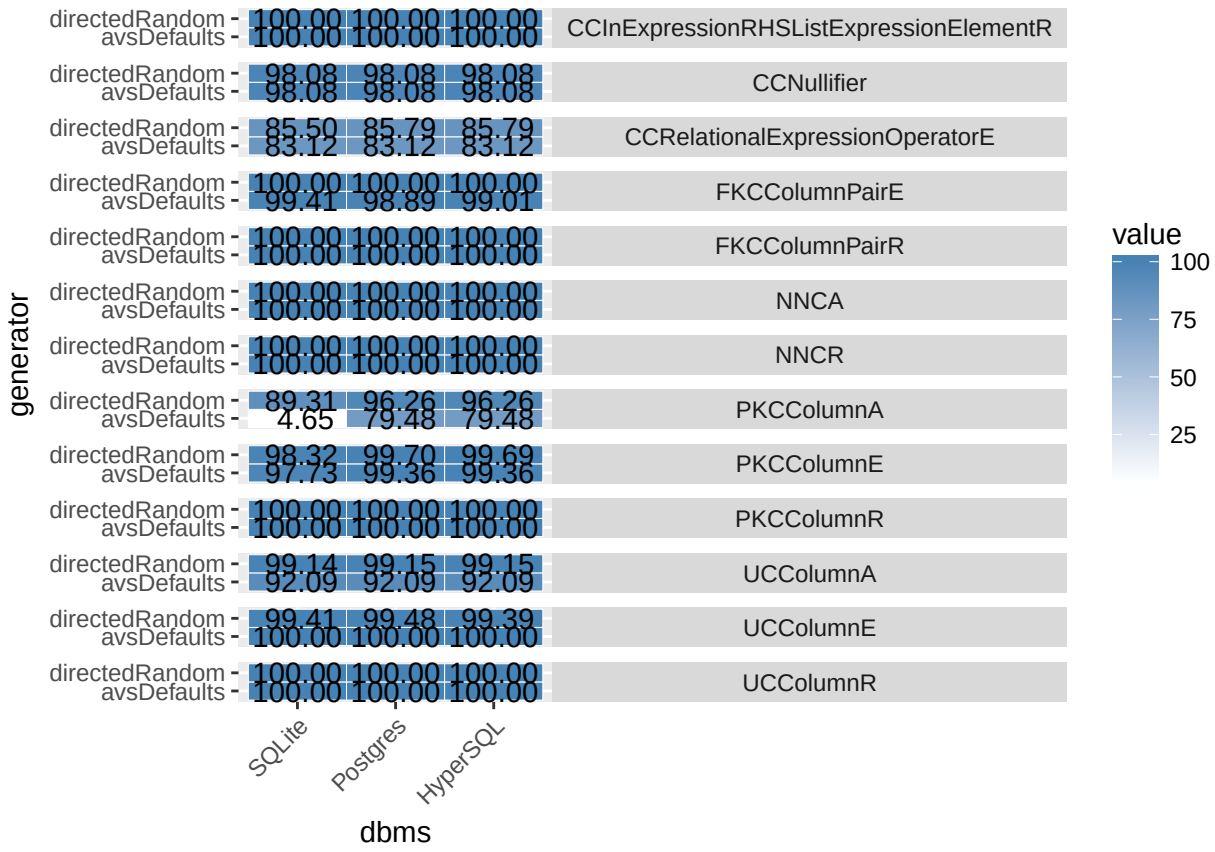

Figure 11: Heat Map of mutant scores in regard of Mutant Operators and DBMSs - in precentage

compared by to DR test suite. Looking at the following figures 12 and 13 showing the summery. And we looked at the PKs especially that uses integers and we found that AVM always try to inset 0 as the only number and for all integer columns except if it is a unique. However, DR randomly generate numbers for integer columns rather than trying to put 0. which I assumed that it might be the reason of AVM having the issue of killing added PKs. However in deeper look and comparing few test suites I found that there is an issue of what is called "composite primary key" which allows for duplication of primary keys.

```
Test requirements covered: 18/18
Coverage: 100.0%
Num Evaluations (test cases only): 49
Num Evaluations (all): 54
Readable Score of TestSuite: 0.5604925665538465
Averge Length of Strings: 5.702127659574468
Number of Empty: 21
JUnit test suite written to generatedtest/Testparsedcasestudy.BankAccount.java
```

Figure 12: Directed Random Bank Account case study for SQLite results

```
Test requirements covered: 18/18
Coverage: 100.0%
Num Evaluations (test cases only): 99
Num Evaluations (all): 101
Readable Score of TestSuite: 0.03161379174429541
Averge Length of Strings: 3.0
Number of Empty: 65
JUnit test suite written to generatedtest/Testparsedcasestudy.BankAccount.java
```

Figure 13: AVM Bank Account case study for SQLite results

Composite primary keys usually called compound key, which consist of have two or more unique keys (mostly primary key) in one table. For example if a table has two columns A and B, which they are both primary key, this table can allow the following insert (1,2) (1,3) (1,4) (2,2) but not (1,2) (2,2). In that sense composite primary keys allows duplication of one key if the other key is unique or vice versa. That is why AVM is losing the battle of not detecting primary key addition operator. AVM might not have the objective function to consider composite primary keys, also DR but it has a random inserts so that is why it detecting high PKColumnA operator (by deep analysis). In the following figures 14, 15 and 16, I show how AVM misses the PKColumnA operator. In those figues I first show the schema that has been mutated by adding "account_name" column in the Account table as PK. In AVM test case the test case passes which does not catch the mutant because that it has all columns as the same data which will throw a exception as intended. However, in DR test case the test generator violate all columns equal and add a different account name in the try-catch which will pass as it is composite key, but the intended behaviour is that it should throw an exception for "id" as it should be unique. As composite keys can have one column duplicated and the other as unique and vice versa, DR caught the mutant (by luck of having a veriaty of generated variables) and AVM did not. Which introduce a new problem for testing databases that we should consider composite key.

Check Constraint Relational Expression Operator Exchange (CCRelationalExpressionOperatorE) which mean changing the relational expressions operators (<, >=, > etc.) within the check constraint. This kind of mutant is hard to kill in some instances however as for DR technique it was slightly higher than AVM, check Figure 11, this is because of DR extracts numbers within the schema and reuse them within the test cases generated. In the other hand, AVM uses default values such as 0 and 1 which in some cases does not test the boundaries of the check constraints.

Unique Column Exchange is another mutant type that exchanges one unique column with another noon-unique column. In Figure 11, AVM shows it has slightly higher score than AVM, and If you check [plots/allcasestudies-DBMSs-HeatMapDiagram.png] it shows that it only happen in BrowserCookies case study. This is because

```
// create the tables for this database
statement.executeUpdate(
    "CREATE TABLE \"UserInfo\" (" +
    "    \"card_number\"    INT    PRIMARY KEY," +
    "    \"pin_number\"    INT    NOT NULL," +
    "    \"user_name\"    VARCHAR(50)    NOT NULL," +
    "    \"acct_lock\"    INT" +
    ")");
statement.executeUpdate(
    "CREATE TABLE \"Account\" (" +
    "    \"id\"    INT," +
    "    \"account_name\"    VARCHAR(50)    NOT NULL," +
    "    \"user_name\"    VARCHAR(50)    NOT NULL," +
    "    \"balance\"    INT," +
    "    \"card_number\"    INT    REFERENCES \"UserInfo\" (\"card_number\")    NOT NULL," +
    "    PRIMARY KEY (\"id\", \"account_name\")" +
    ")");
```

Figure 14: Schema Initialisation in Junit with “account_name” column added as PK

```
// 3-Account: UNIQUE[id] for Account - all cols equal
// 14-Account: id is NOT UNIQUE
// (¬Null[Account: account_name] ∧ ¬Null[Account: user_name] ∧ ¬Null[Account: card_number] ∧ (Match[=[Account: card
// Result is: false

// prepare the database state
assertEquals(1, statement.executeUpdate(
    "INSERT INTO \"UserInfo\"(" +
    "    \"card_number\", \"pin_number\", \"user_name\", \"acct_lock\"" +
    ") VALUES (" +
    "    0, 0, \"\", 0" +
    ")"));
assertEquals(1, statement.executeUpdate(
    "INSERT INTO \"Account\"(" +
    "    \"id\", \"account_name\", \"user_name\", \"balance\", \"card_number\"" +
    ") VALUES (" +
    "    0, \"\", \"\", 0, 0" +
    ")"));

// execute INSERT statements for the test case
try {
    statement.executeUpdate(
        "INSERT INTO \"Account\"(" +
        "    \"id\", \"account_name\", \"user_name\", \"balance\", \"card_number\"" +
        ") VALUES (" +
        "    0, \"\", \"\", 0, 0" +
        ")");
    fail("Expected constraint violation did not occur");
} catch (SQLException e) { /* expected exception thrown and caught */ }
```

Figure 15: AVM Test case that does not catch the mutant

```
// 3-Account: UNIQUE[id] for Account - all cols equal
// 14-Account: id is NOT UNIQUE
// (¬Null[Account: account_name] ∧ ¬Null[Account: user_name] ∧ ¬Null[Account: card_number] ∧ (Match[=[Account:
// Result is: false

// prepare the database state
assertEquals(1, statement.executeUpdate(
    "INSERT INTO \"UserInfo\"(" +
    "    \"card_number\", \"pin_number\", \"user_name\", \"acct_lock\"" +
    ") VALUES (" +
    "    -585, -988, \"wm\", -262" +
    ")"));
assertEquals(1, statement.executeUpdate(
    "INSERT INTO \"Account\"(" +
    "    \"id\", \"account_name\", \"user_name\", \"balance\", \"card_number\"" +
    ") VALUES (" +
    "    166, \"\", \"ruan\", -37, -585" +
    ")"));

// execute INSERT statements for the test case
try {
    statement.executeUpdate(
        "INSERT INTO \"Account\"(" +
        "    \"id\", \"account_name\", \"user_name\", \"balance\", \"card_number\"" +
        ") VALUES (" +
        "    166, \"qjwhukbdo\", \"pmcpao\", NULL, -585" +
        ")");
    fail("Expected constraint violation did not occur");
} catch (SQLException e) { /* expected exception thrown and caught */ }
```

Figure 16: DR Test case that catches the mutant and kill it.

one table has three composite unique keys which this type of unique key exchange one column with one that is not unique. As AVM uses default values, for performance reasons, it tries to insert a row that is similar to the T-suffiecent data, which leads to breaks the unique composite keys and it kills the mutant (failed test). However in DR the variables generated are not similar so the test will pass with no issue of breaking the unique composite keys, please view the followng figures 17 and 18 two see the differences of Unique Composite Keys.

Analyse Wilcox Rank and Effect Size for AVM and Directed Random

To statistically analyze the the new technique we conducted tests for significance with the nonparametric Wilcoxon rank-sum test, using the sets of 30 execution times obtained with a specific DBMS and all techniques **Hitchhicker Guide Ref**. A p-value that less than 0.05 is considered significant. To review practical are significance tests we use the nonparametric A12 statistic of Vargha and Delaney **REF** was used to calculate effect sizes. The A12 determine the average probability that one approach beats another, or how superior one technique compared to the other. We followed the guidelines of Vargha and Delaney in that an effect size is considered to be "large" if the value of A12 is $ < 0:29$ or > 0.71, "medium" if A12 is < 0.36 or > 0.64 and "small" if A12 is < 0.44 or > 0.56. However, is the values of A12 close to the 0.5 value are viewed as there no effect.

When comparing AVM and Directed Random techniques in regard of time we used Mann-Whitney U-test and the A-hat effect size calculations. As Shown in in the following two tables that there is statistically significat differance between Driected Random and AVM, $p \leq 0.05$. Therefore, we reject the null hypothesis that there is no difference between AVM and Directed Random. As p-value near zero, that directed random is faster than AVM in a statistically significant test. Moreover, the A-12 shows that Directed Random has a large effect size when it comes to test generation timing. Which menas that Directed Random is the winner in regard of test generation timing.

p.value	dbms	vs
0.93791	Postgres	AVM vs Directed Random Coverage
1.00000	SQLite	AVM vs Directed Random Coverage
1.00000	HyperSQL	AVM vs Directed Random Coverage
0.00000	Postgres	AVM vs Directed Random Mutation Score

p.value	dbms	vs
0.00000	SQLite	AVM vs Directed Random Mutation Score
0.00000	HyperSQL	AVM vs Directed Random Mutation Score
0.00000	Postgres	AVM vs Directed Random Number Of evaluations
0.00000	SQLite	AVM vs Directed Random Number Of evaluations
0.00000	HyperSQL	AVM vs Directed Random Number Of evaluations
0.00000	Postgres	AVM vs Directed Random Test Generation Time
0.00000	SQLite	AVM vs Directed Random Test Generation Time
0.00000	HyperSQL	AVM vs Directed Random Test Generation Time

HeatMap Plot of mutant analysis per technique for each Operator

```
CREATE TABLE "cookies" (
    "id"    INT PRIMARY KEY NOT NULL,
    "name"  TEXT    NOT NULL,
    "value" TEXT,
    "expiry"    INT,
    "last_accessed" INT,
    "creation_time" INT,
    "host"  TEXT,
    "path"  TEXT,
    FOREIGN KEY ("host", "path") REFERENCES "places" ("host", "path"),
    UNIQUE ("name", "host", "path"),
    CHECK ("expiry" = 0 OR "expiry" > "last_accessed"),
    CHECK ("last_accessed" >= "creation_time")
)

        |
        |
        |
        V

CREATE TABLE "cookies" (
    "id"    INT PRIMARY KEY NOT NULL,
    "name"  TEXT    NOT NULL,
    "value" TEXT,
    "expiry"    INT,
    "last_accessed" INT,
    "creation_time" INT,
    "host"  TEXT,
    "path"  TEXT,
    FOREIGN KEY ("host", "path") REFERENCES "places" ("host", "path"),
    UNIQUE ("creation_time", "host", "path"),
    CHECK ("expiry" = 0 OR "expiry" > "last_accessed"),
    CHECK ("last_accessed" >= "creation_time")
)
```

Figure 17: Image of the mutant that DR did not kill but AVM killed

```java
public void test24() throws SQLException {
    // 30-cookies: name is UNIQUE
    // ((Match[≠[cookies: id]] ∨ Null[cookies: id]) ∧ ¬Null[cookies: id] ∧ ¬Null[cookies: name] ∧ (vMatch[≠[cookies: name
    // Result is: true

    // prepare the database state
    assertEquals(1, statement.executeUpdate(
        "INSERT INTO \"places\"(" +
        "    \"host\", \"path\", \"title\", \"visit_count\", \"fav_icon_url\"" +
        ") VALUES (" +
        "    '', '', '', 0, ''" +
        ")"));
    assertEquals(1, statement.executeUpdate(
        "INSERT INTO \"cookies\"(" +
        "    \"id\", \"name\", \"value\", \"expiry\", \"last_accessed\", \"creation_time\", \"host\", \"path\"" +
        ") VALUES (" +
        "    0, '', '', 0, 0, 0, '', ''" +  // the last three are unique of (0, '', '')
        ")"));

    // execute INSERT statements for the test case
    assertEquals(1, statement.executeUpdate(
        "INSERT INTO \"places\"(" +
        "    \"host\", \"path\", \"title\", \"visit_count\", \"fav_icon_url\"" +
        ") VALUES (" +
        "    'a', '', '', 0, ''" +
        ")"));
    assertEquals(1, statement.executeUpdate(
        "INSERT INTO \"cookies\"(" +
        "    \"id\", \"name\", \"value\", \"expiry\", \"last_accessed\", \"creation_time\", \"host\", \"path\"" +
        ") VALUES (" +
        "    1, 'a', '', 0, 0, 0, '', ''" +   // Same last three entries as before (0, '', '') which leads to failure
        ")"));
}
```

Figure 18: AVM test case that killed the mutant because of uniques of all three column.

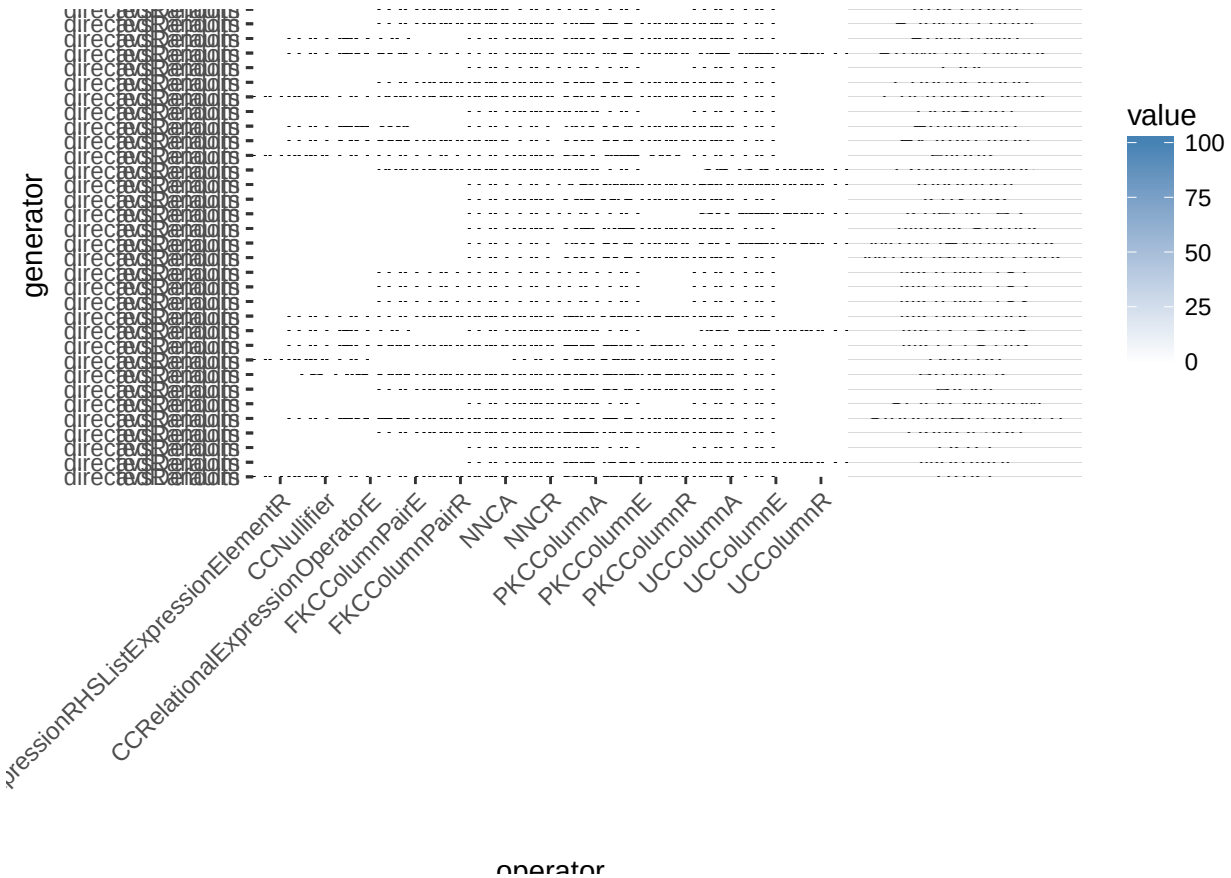

Figure 19: Heat Map of mutant scores in regard of Mutant Operators and DBMSs - in precentage