Driected Random

Abdullah Alsharif

Directed Random Mechanism

Directed Random works as same as random technique however it is guided by predicates that are checked then used to fix the random generated 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. 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. # 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 ?

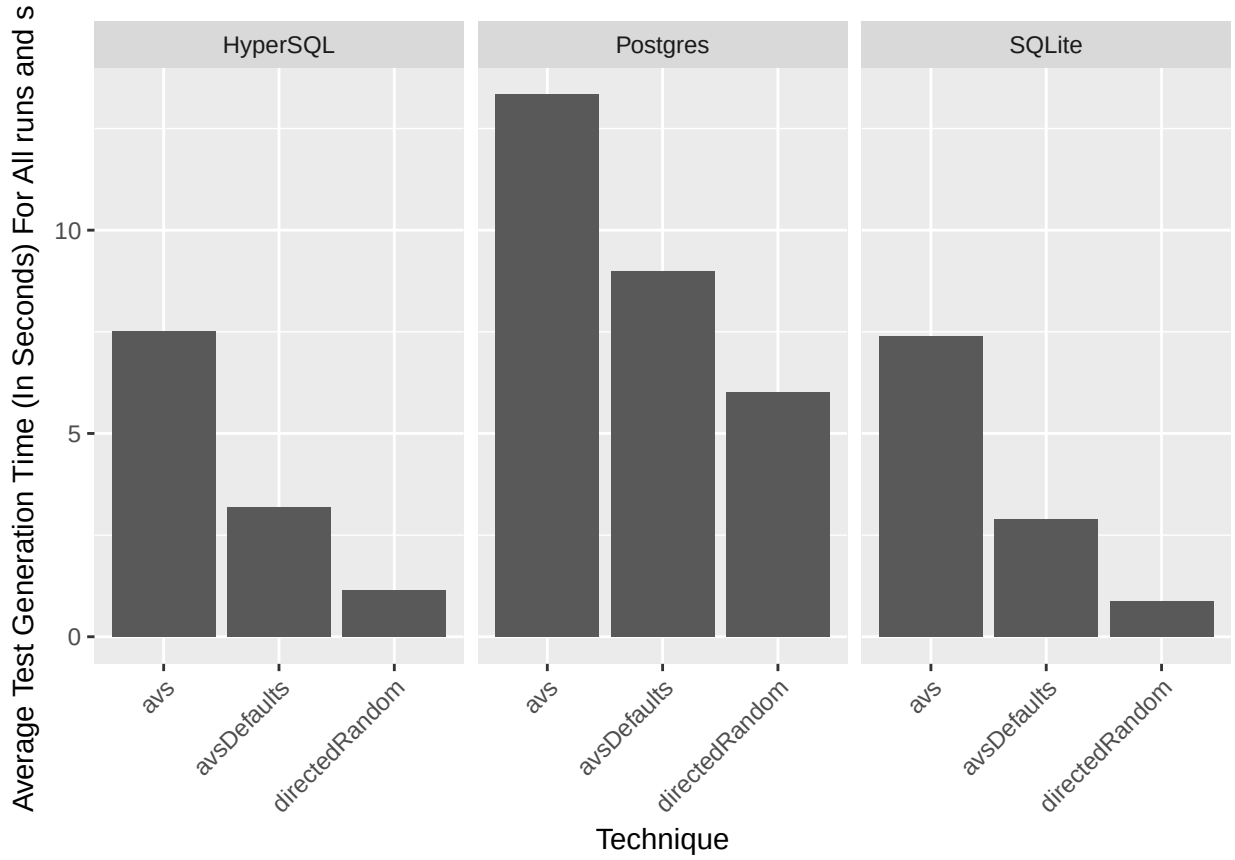

Figure 1: Avrages of Test generation timing - in seconds. Group By data generator and DBMS, then average test generation timing and then when ploting divide by 1000 to convert to seconds

To look in more details we split test generation timing analysis for each case study. In Figure 2, I 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

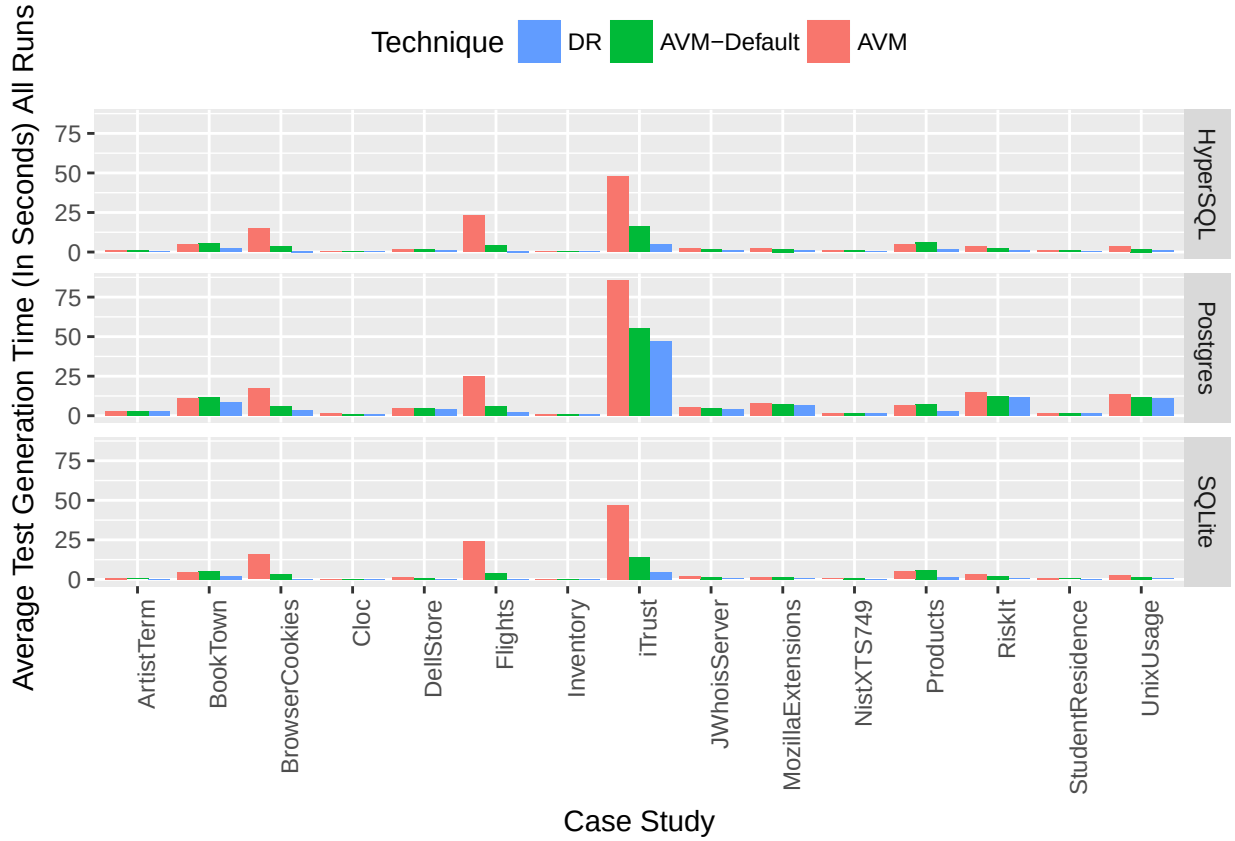

Figure 2: Avrages of Test generation timing for each schema- in seconds. Group by data generator, DBMS and case sties, average test generation timing, then divide by 1000 to convert to second.

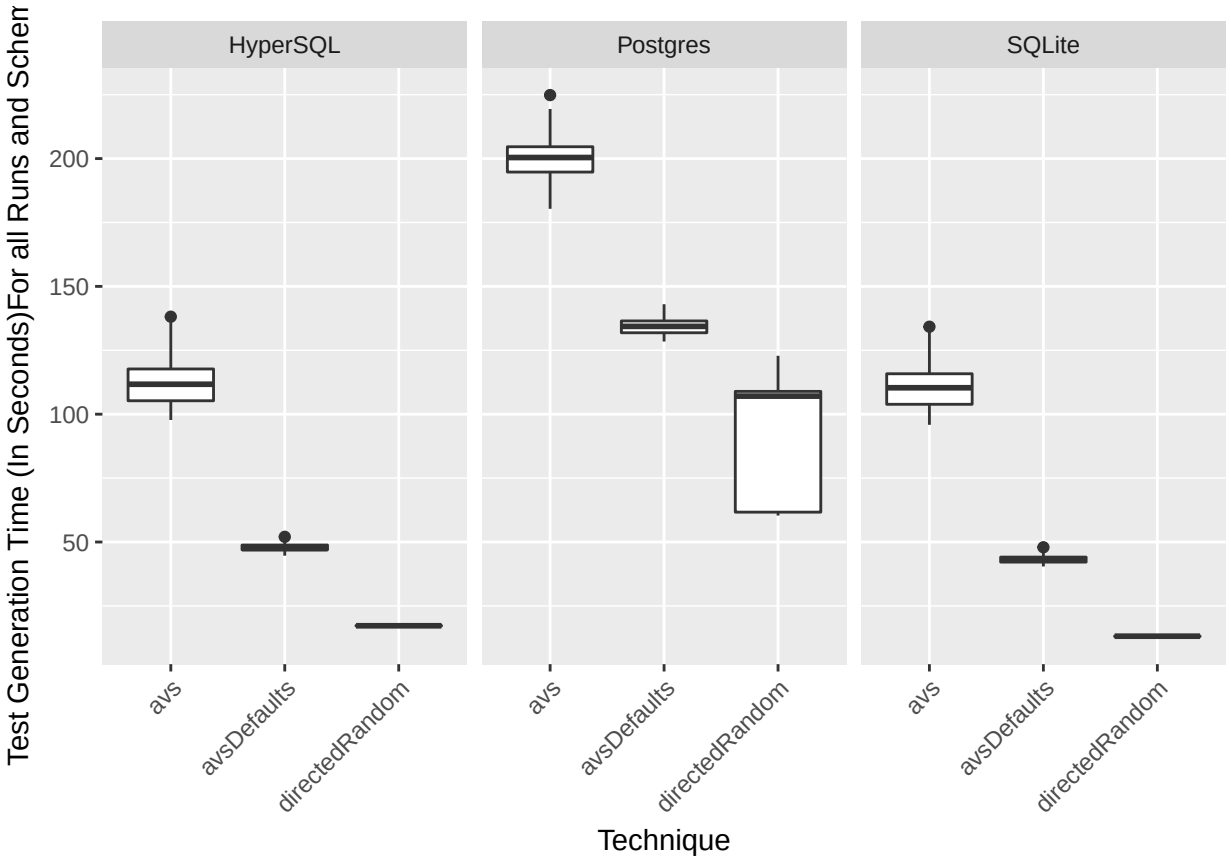

Figure 3: Test generation timing - in seconds. First summing test generation timing for each run then group by data generator and DBMS, then divide test generation timing by 1000 to convert to second.

test cases.

Coverages

Mutation Scores

In Figure 6, 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.

```
## NULL
```

In Figure 7, 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 8, I show the spread of values of mutation score in regard of DBMS and technique using a box plot, for all runs and schemas.

In Figure 9, I show the spread of values for mutation scores for each schema, DBMS and technique, for all runs.

Mutation Scores and Mutation Operators

In Figure 10, 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 11 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 12 We show that the average percentages of all schemas for each operator that is been killed for each technique. The figure shows that DR always has 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 compared by to DR test suite. Looking at the following figures 13 and 14, showing the summery of empty string within each test suite. I also 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.

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,

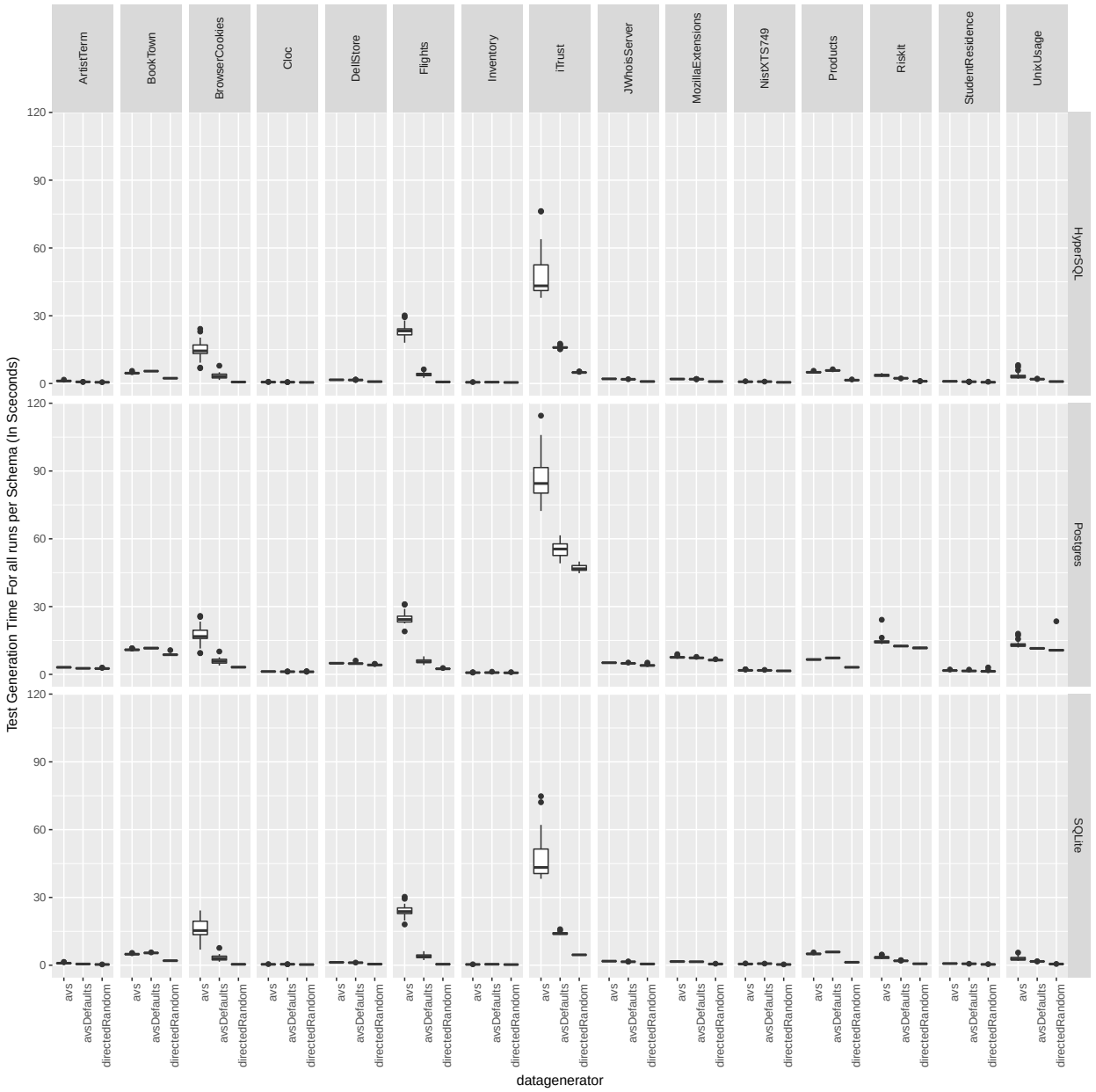

Figure 4: Box plot for Test Generation time for DBMS, techniques and schemas - in precentage. Group by data generator, case study and DBMS, then dividing test generation timning by 1000 to convert to second. No averaging or summing to see the spread of values for all runs per schema

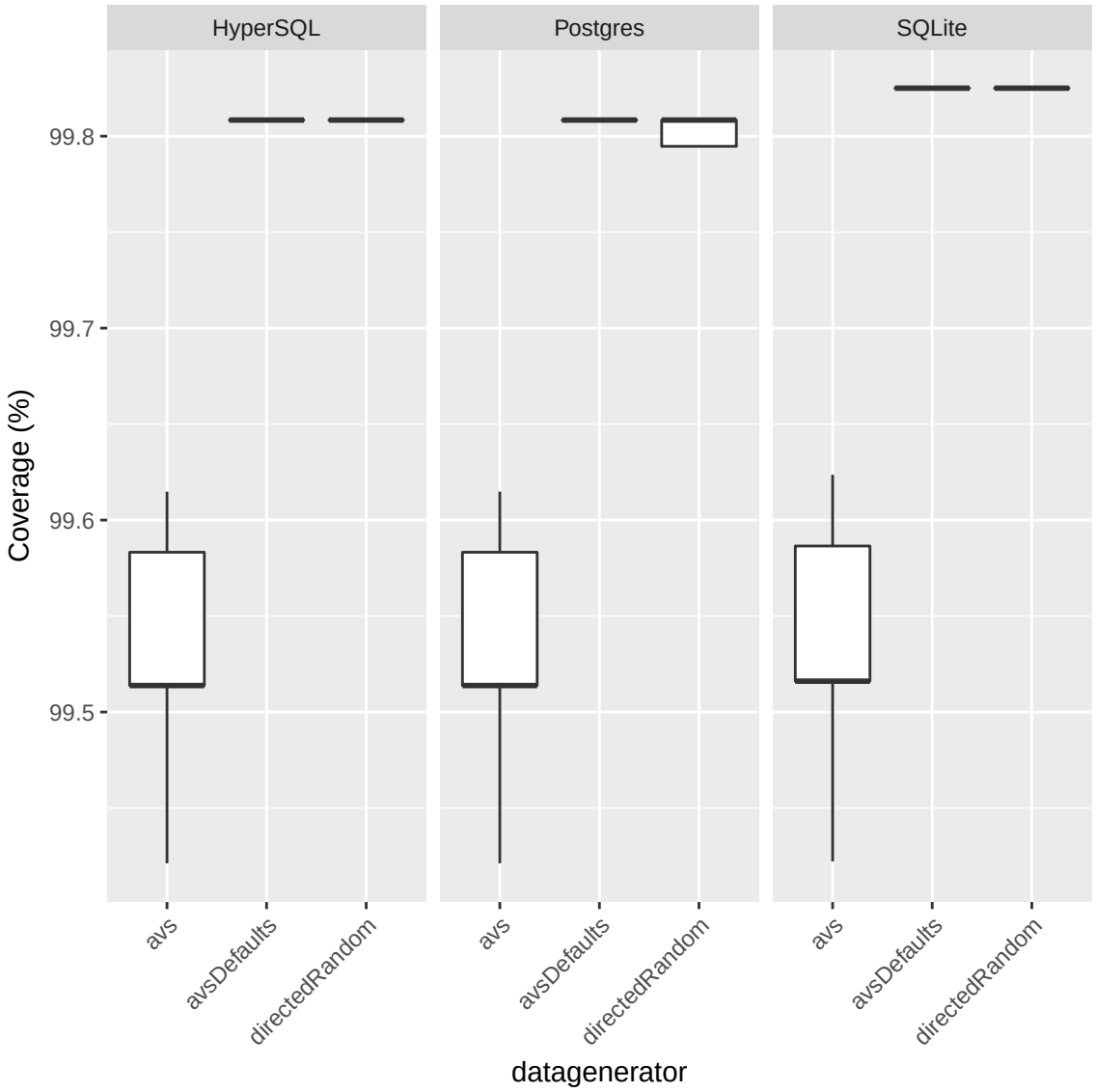

Figure 5: Box plot for Coverage, averaging all schema coverage per run

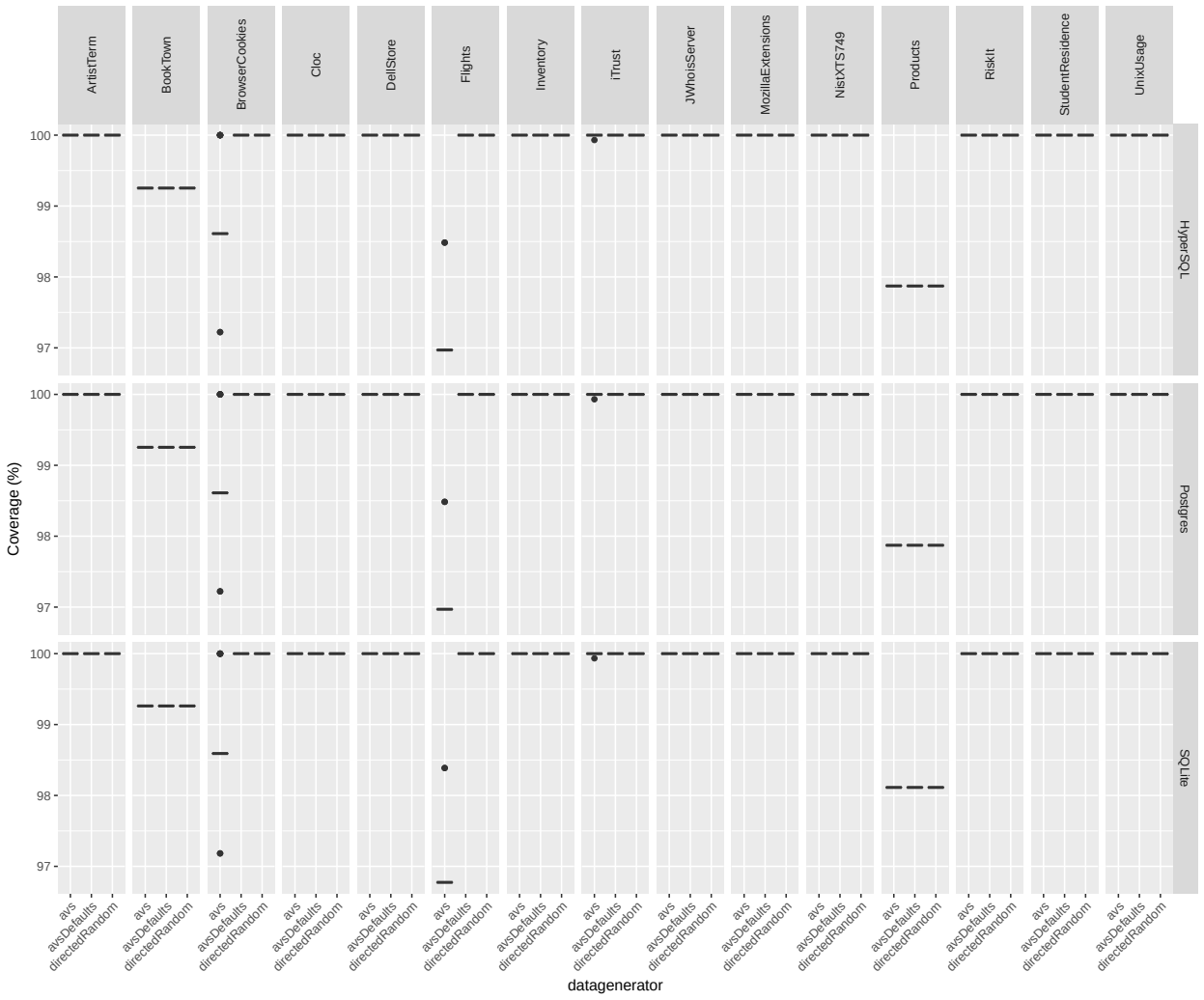

Figure 6: Box plot for Coverage

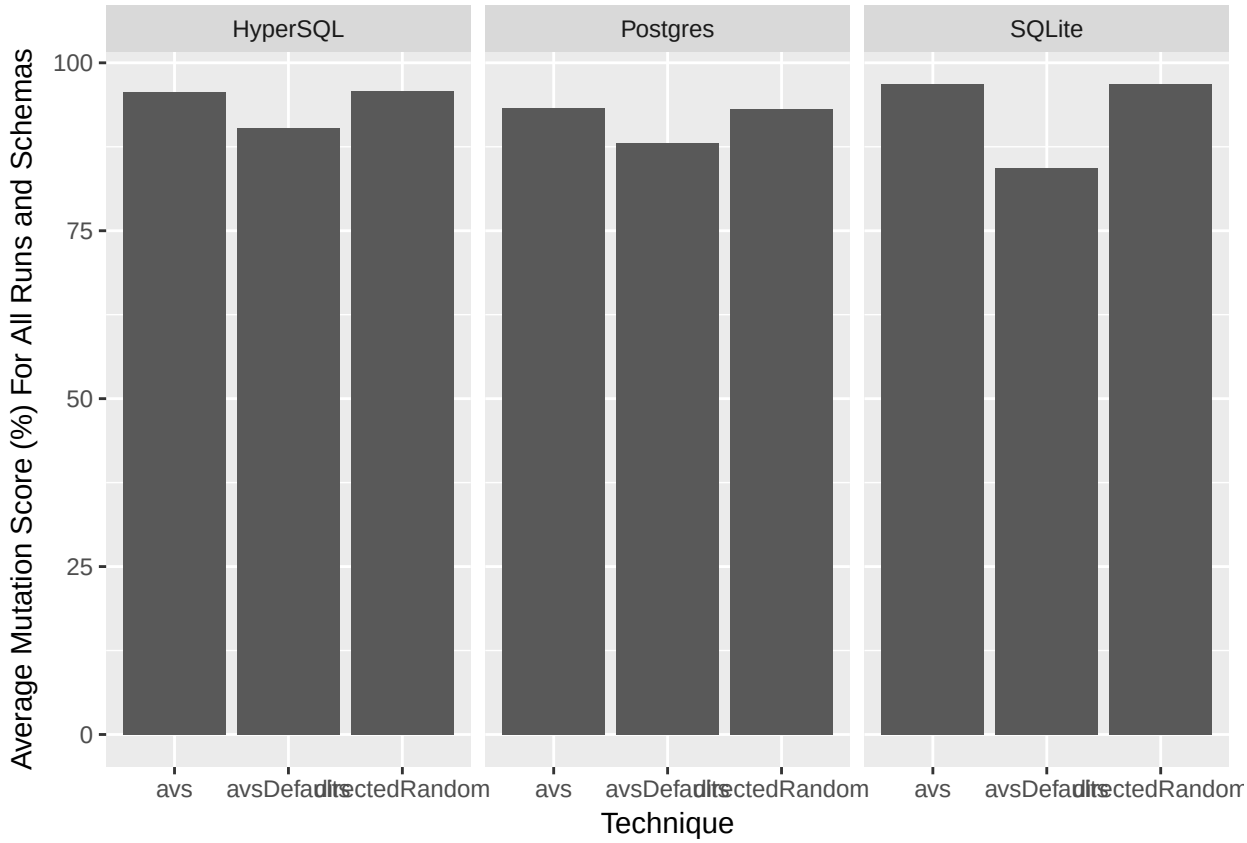

Figure 7: Avrages of Mutation Score - in precentage. Grouping by data generator and DBMS, averaging score numerator and denominator for all runs and schemas, then ploting it by divding the numerator by denominator multiplying by 100

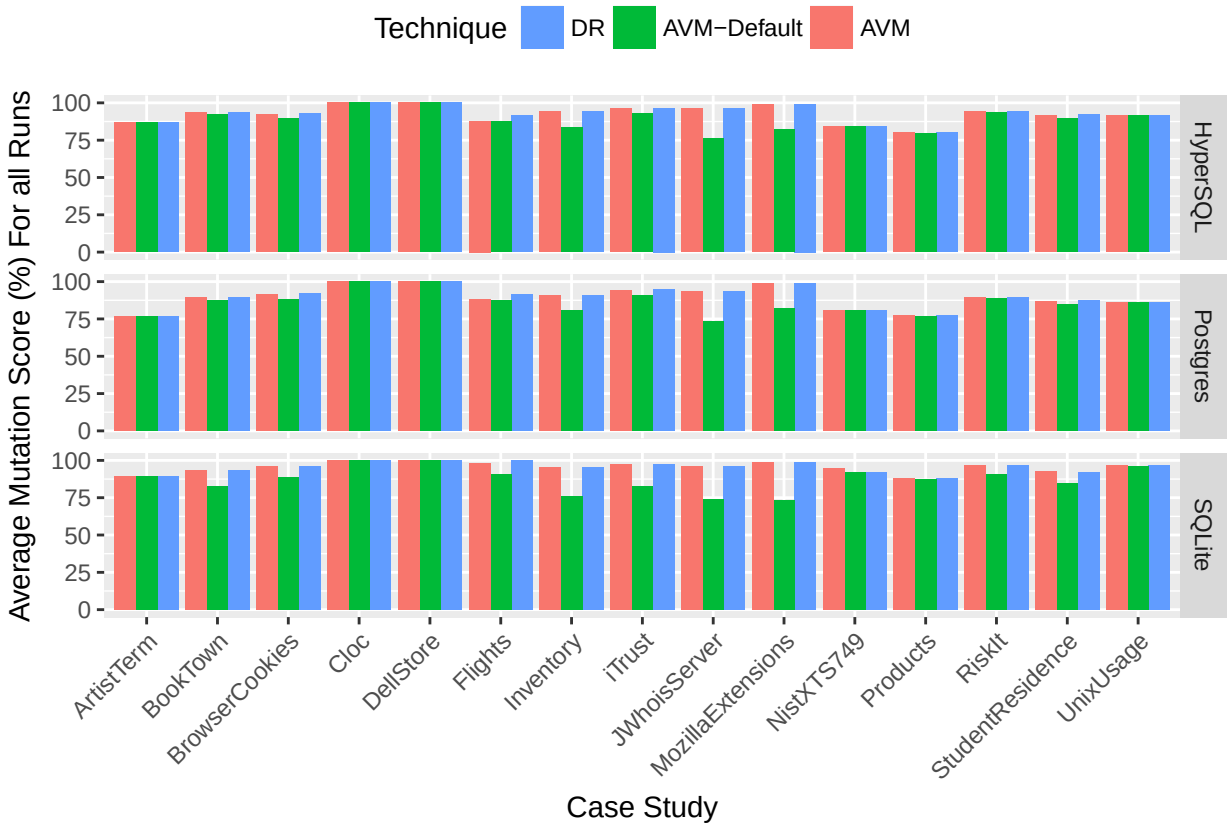

Figure 8: Avrages of Mutation Score for each schema - in precentage. Group by data generator, DBMS and case studies, then averaging score numerator and denominator for all runs per schema (as been grouped by it), then ploting it by divding the numerator by denominator multiplying by 100

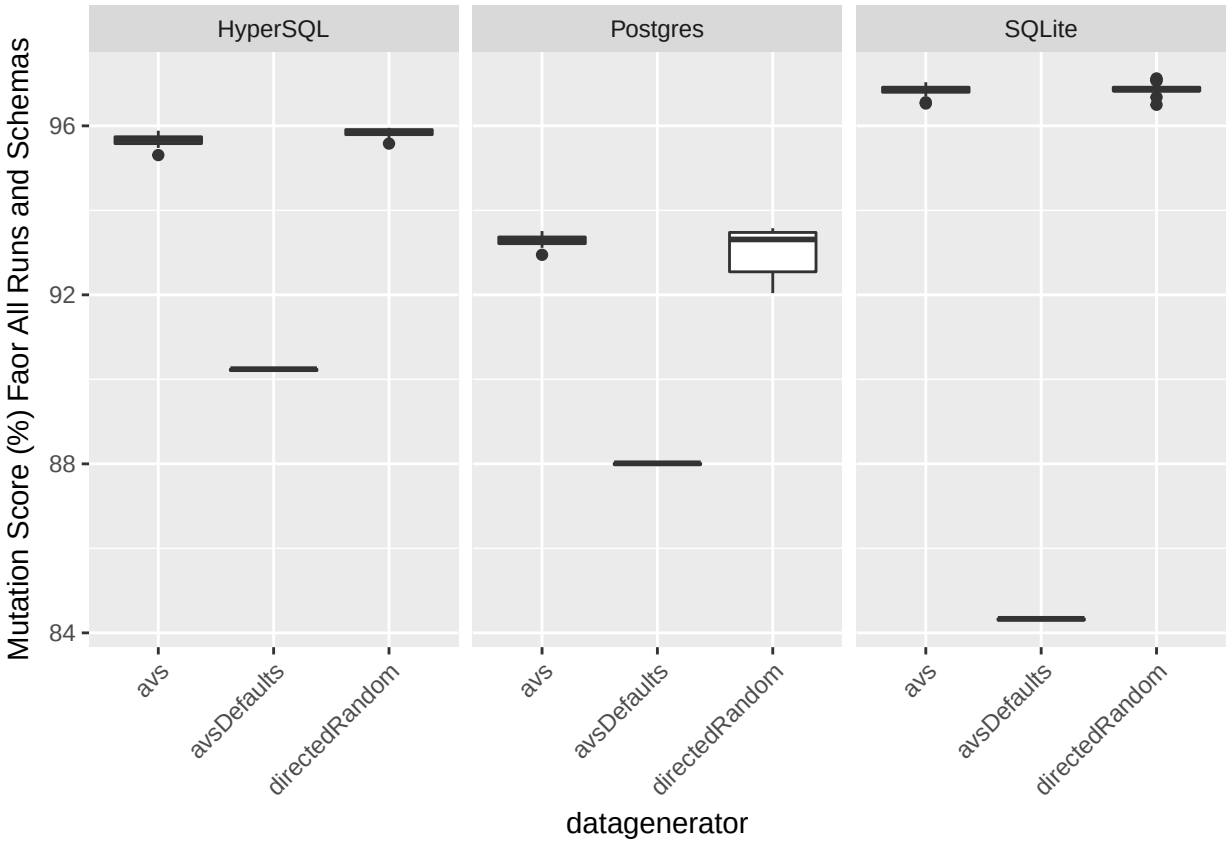

Figure 9: Box plot for mutation scores for DBMSs and techniques - in precentage. Frist I sum all score numerator and denominator per run per DBMS per data generator, then plottig using group by data generator, DBMS and divding the score numerator by denominator multiplying by 100

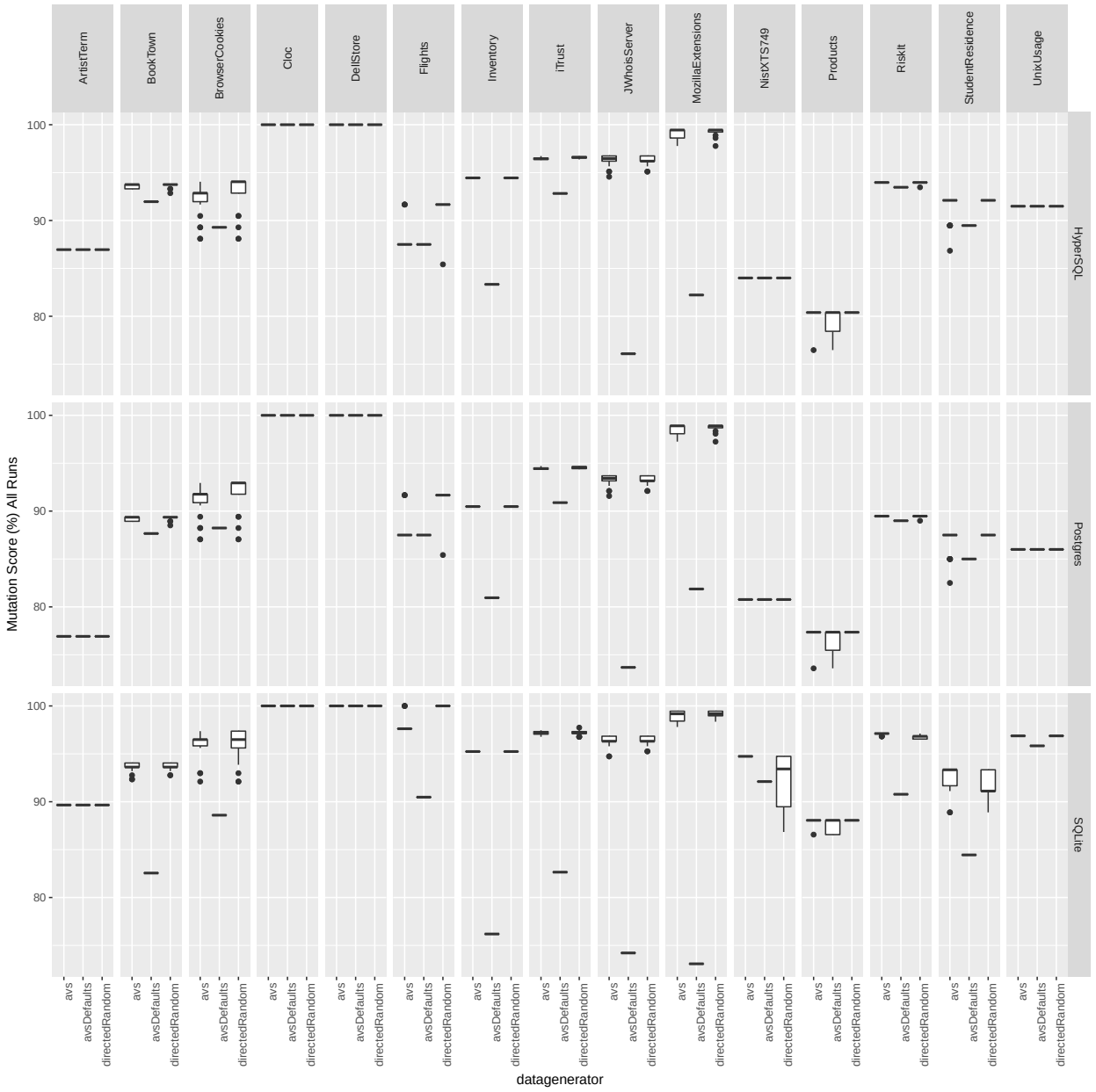

Figure 10: Box plot for mutation scores for DBMS, techniques and schemas - in precentage. Group by data generator, case study and DBMS, then divding the score numerator by denominator multiplying by 100. No averaging or summing to see the spread of values for all runs per schema

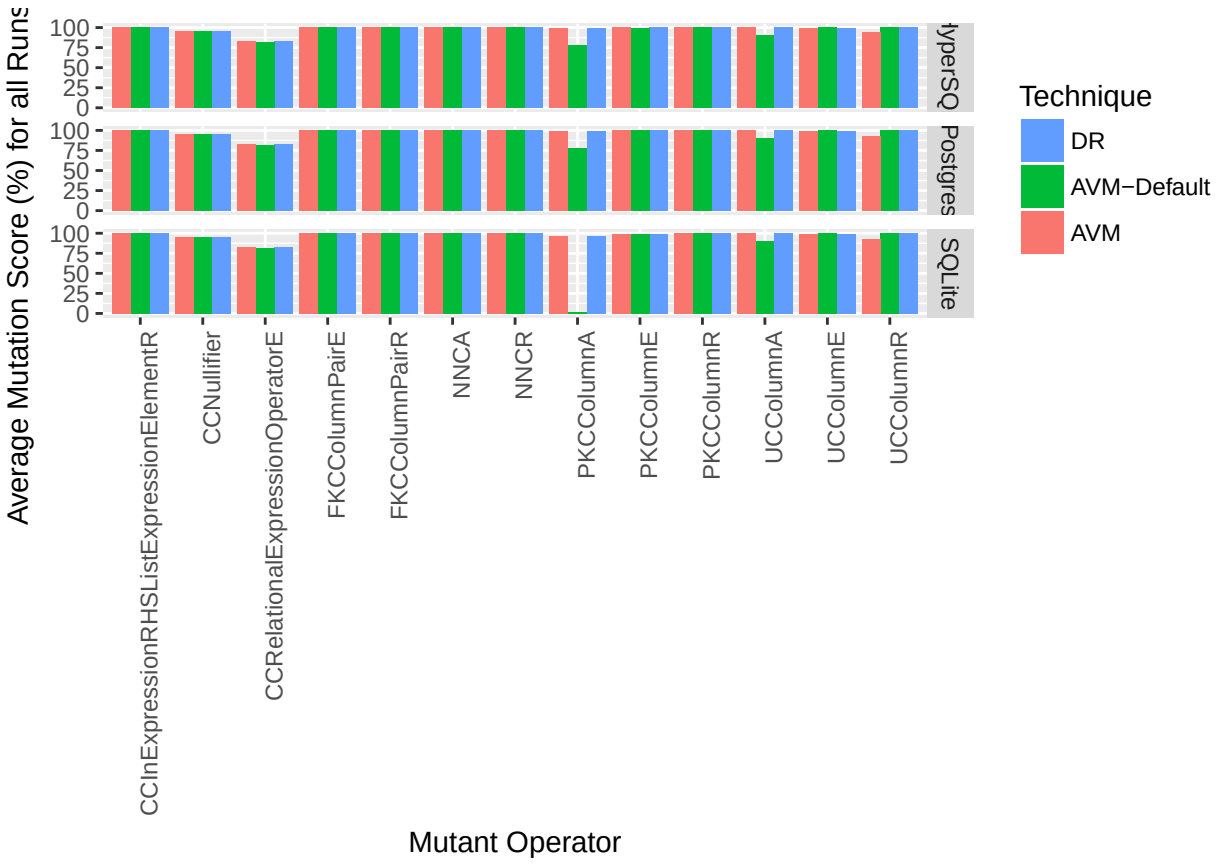

Figure 11: Averages of mutant scores in regard of Mutant Operators and DBMSs - in precentage. Using mutanttiming file, I group by generator, DBMS and operator (FIXED by removing grouping by schema). Then I only select NORMAL mutants, then calculating the percentage of killed mutant by summing all killed divided by (killed mutant plus alive mutant) multiply by 100.

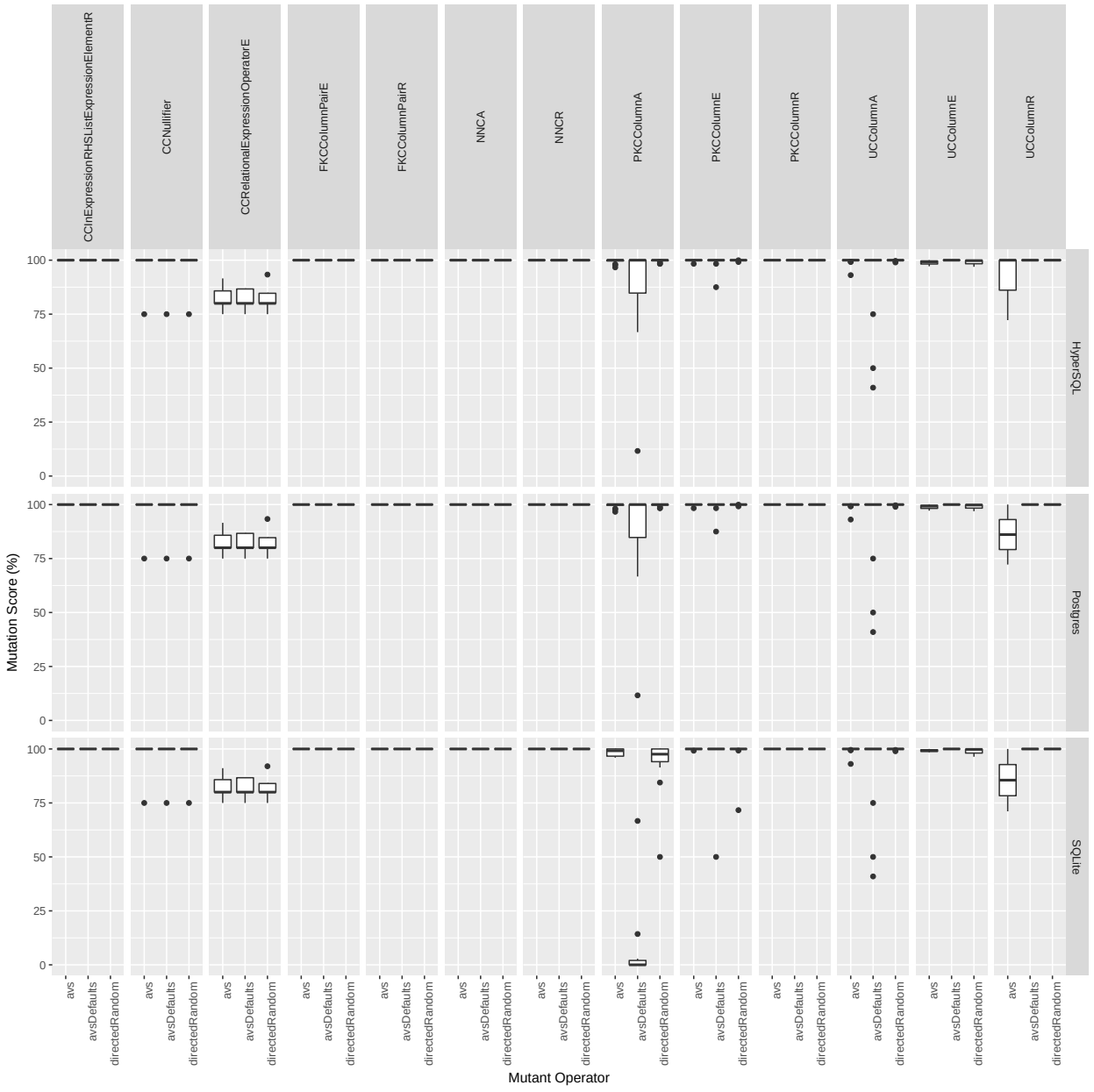

Figure 12: Box Plot of mutant scores in regard of Mutant Operators and DBMSs - in precentage. Using mutanttiming file, I group by generator, DBMS, schema and operator. Then I only select NORMAL mutants, then calculating the percentage of killed mutant by summing all killed divided by (killed mutant plus alive mutant) multiply by 100.

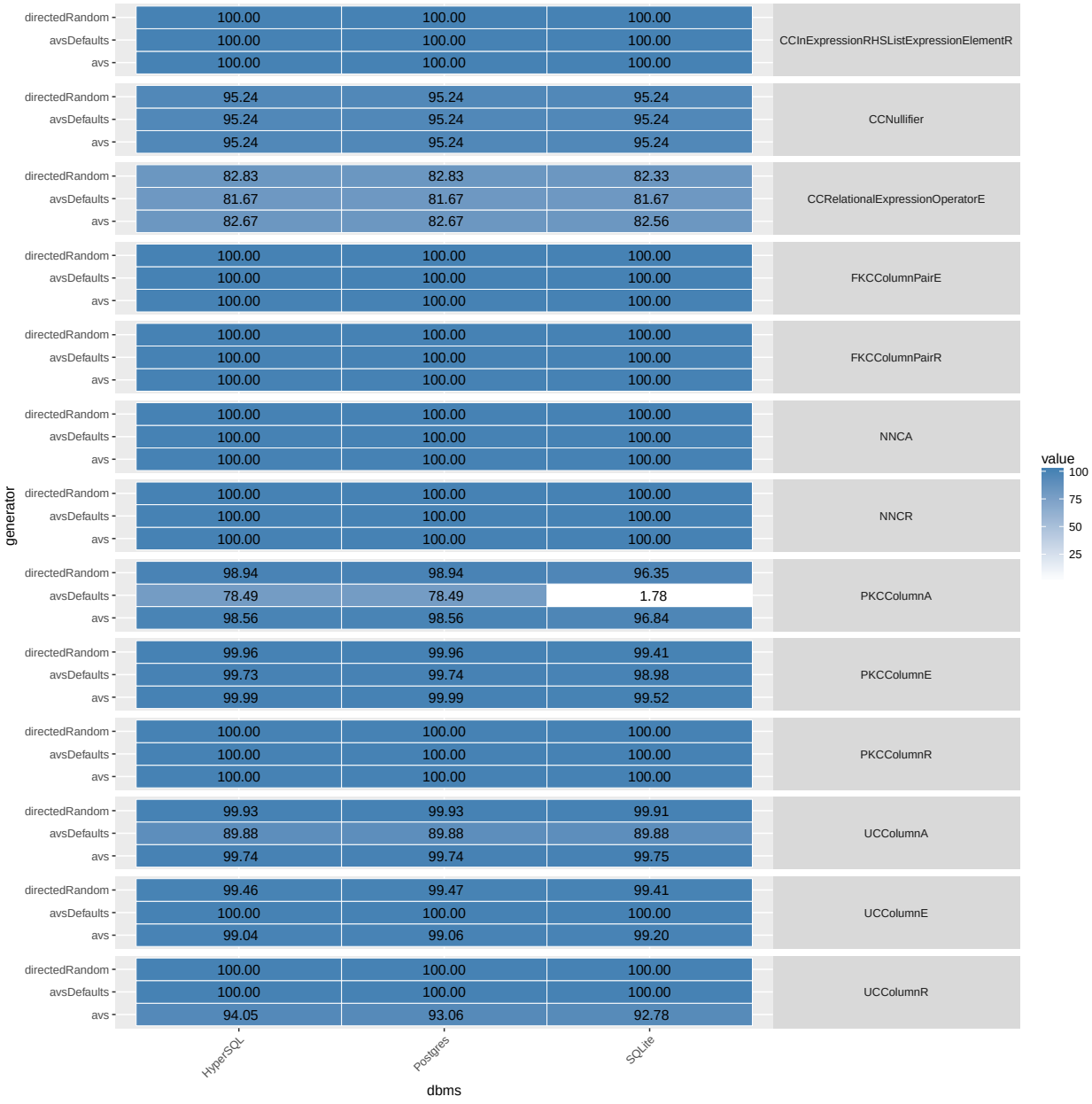

Figure 13: Heat Map of mutant scores in regard of Mutant Operators and DBMSs - in percentage. Using mutanttiming file after selecting mutant that has type NORMAL, summing all killed mutant and total mutants (Killed + Alive) grouped by generator, schema, operator and DBMS. Then plotting by grouping the previous calculated data by DBMS, generator and operator, then got the percentage of killed mutants per DBMS, generator and DBMS.

```
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 14: 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 15: AVM Bank Account case study for SQLite results

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. In the following figures 15, 16 and 17, 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 cases passes which does not catch the mutant because all the PK columns have the same values as the T-sufficient values which will throw a exception as intended. However, in DR test cases violate all the PK columns beucase account name has a different value than T-sufficient values, which will be inserted in the database, and the try-catch will fail as there is no errors. 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 becuase of the default values.

```
// 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 16: Schema Initialisation in Junit with "account_name" column added as PK

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

```
// 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 17: 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 18: DR Test case that catches the mutant and kill it.

boundaries of the check constraints.

Unique Column Exchange is another mutant type that exchanges one unique column with another non-unique column. In Figure 12, AVM shows it has slightly higher score than AVM, and If you check Figure 19 it shows that it only happen in BrowserCookies case study. This is because one table has three composite unique keys, which leads the mutation to exchange one of the three columns with a non-unique cloumn. As AVM uses default values, for performance reasons, it tries to insert a row that is similar to the T-suffiecent value, which leads to breaks the unique composite keys, and the test will not pass, meaning it kills the mutant. However in DR had different values generated that are not similar so the test will pass with and the data will be inserted within the database, no issue of breaking the unique composite keys, please view the followng figures 18 and 19 two see the differences of Unique Composite Keys.

```
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 19: Image of the mutant that DR did not kill but AVM killed

```
public void test24() throws SQLException {
    // 30-cookies: name is UNIQUE
    // ((Match[≠[cookies: id]] ∨ Null[cookies: id]) ∧ ¬Null[cookies: id] ∧ ¬Null[cookies: name] ∧ (∨Match[≠[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 20: AVM test case that killed the mutant because of uniques of all three column.

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
0.8529419	Postgres	AVM vs Directed Random Coverage
1.0000000	SQLite	AVM vs Directed Random Coverage
1.0000000	HyperSQL	AVM vs Directed Random Coverage
0.0000000	Postgres	AVM vs Directed Random Mutation Score
0.0000000	SQLite	AVM vs Directed Random Mutation Score
0.0000000	HyperSQL	AVM vs Directed Random Mutation Score
0.0000000	Postgres	AVM vs Directed Random Number Of evaluations
0.0000000	SQLite	AVM vs Directed Random Number Of evaluations
0.0000000	HyperSQL	AVM vs Directed Random Number Of evaluations
0.0000000	Postgres	AVM vs Directed Random Test Generation Time

p.value	dbms	vs
0.0000000	SQLite	AVM vs Directed Random Test Generation Time
0.0000000	HyperSQL	AVM vs Directed Random Test Generation Time

HeatMap Plot of mutant analysis per technique for each Operator

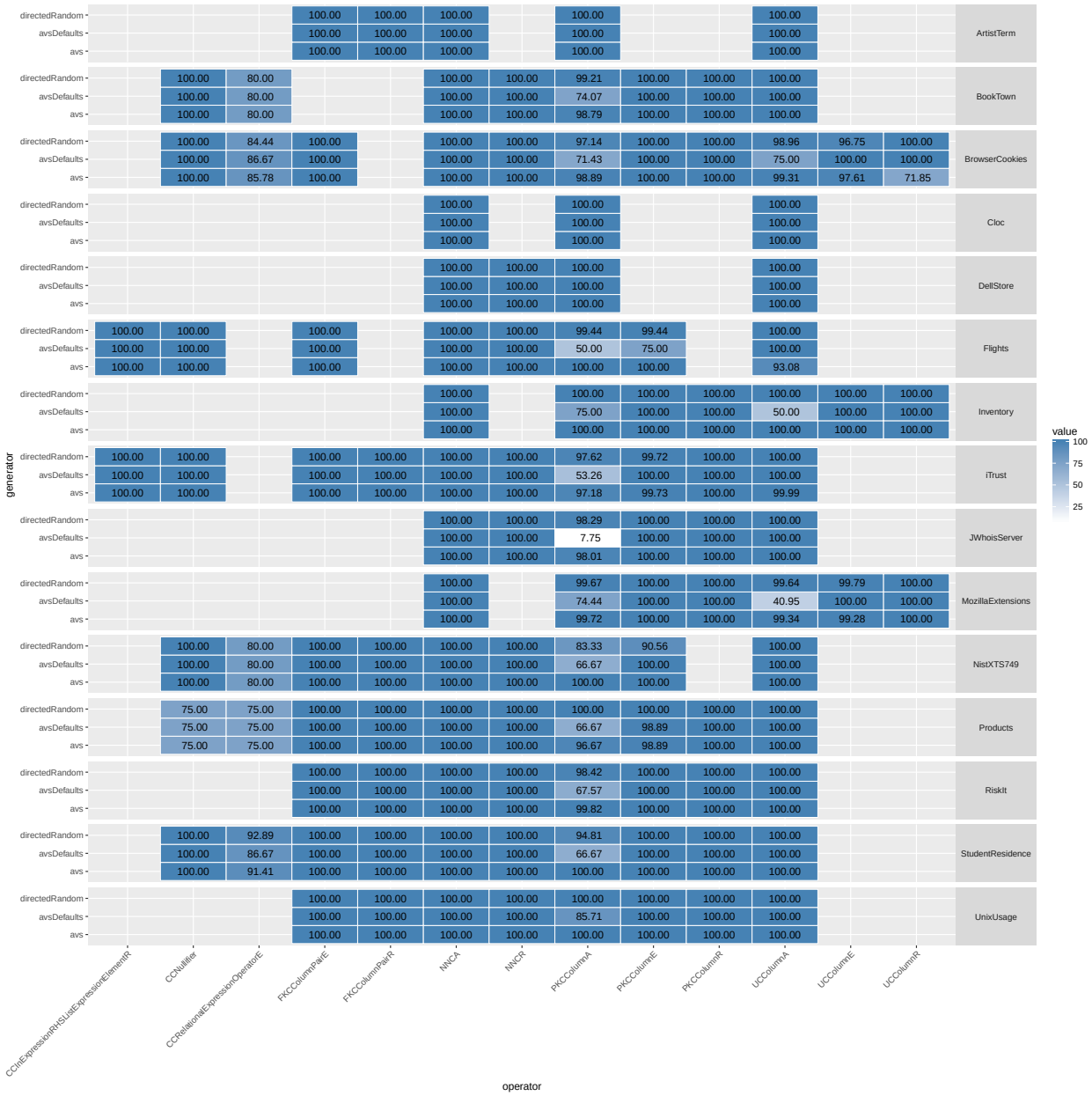

Figure 21: Heat Map of mutant scores in regard of Mutant Operators for each schema per generator - in precentage

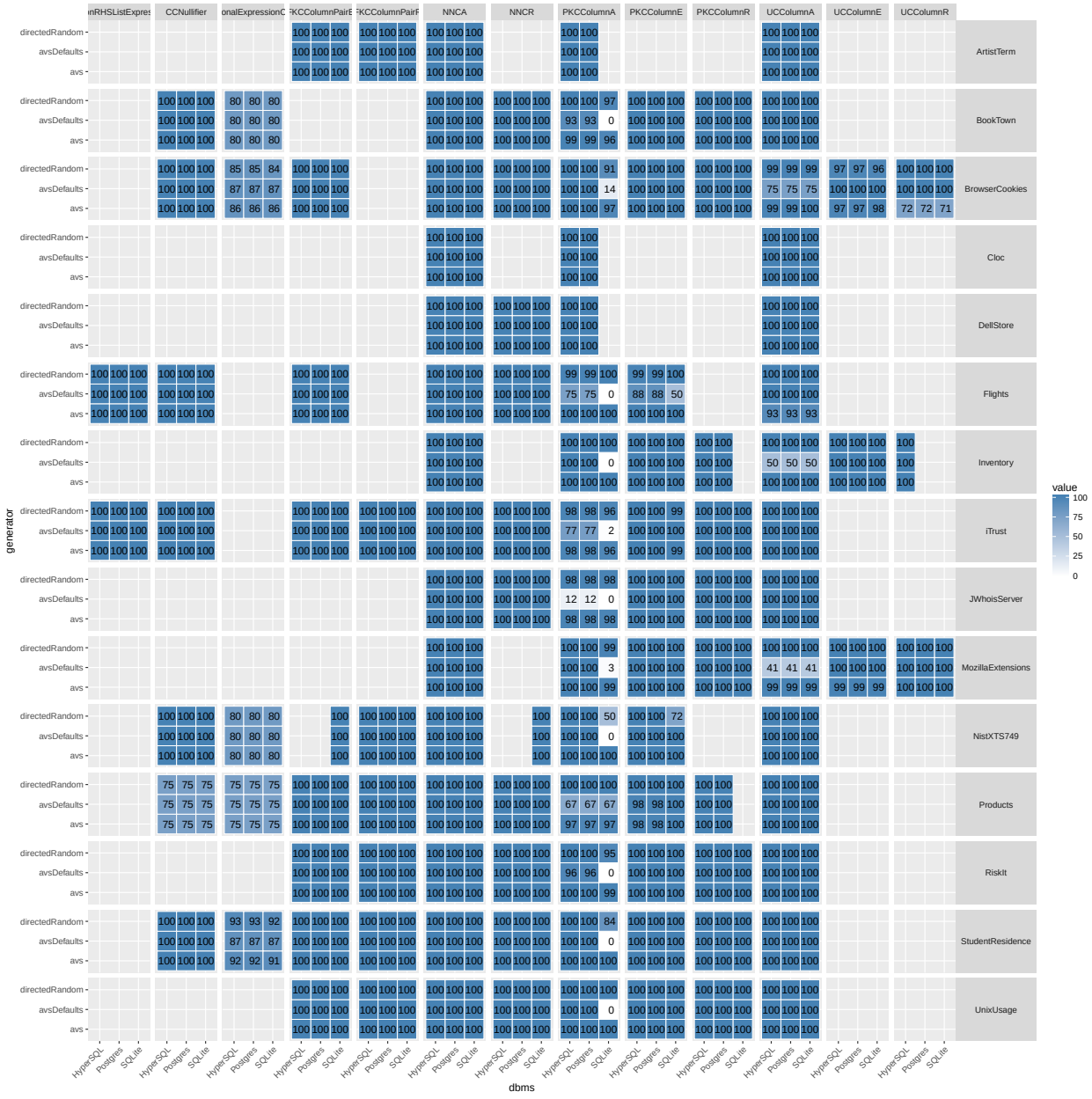

Figure 22: Detailed Heat Map of mutant scores in regard of Mutant Operators and DBMSs per schema - in precentage