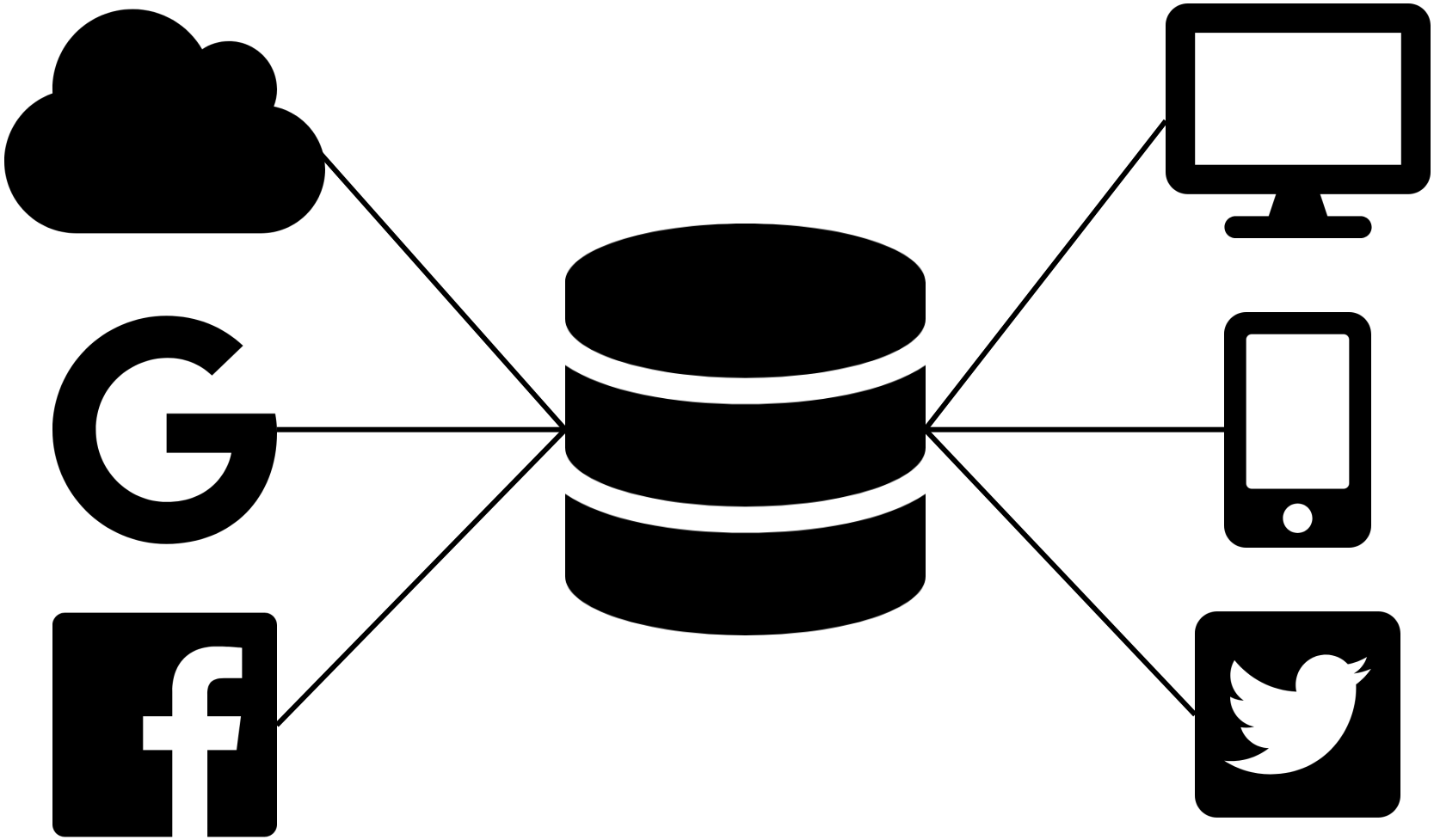
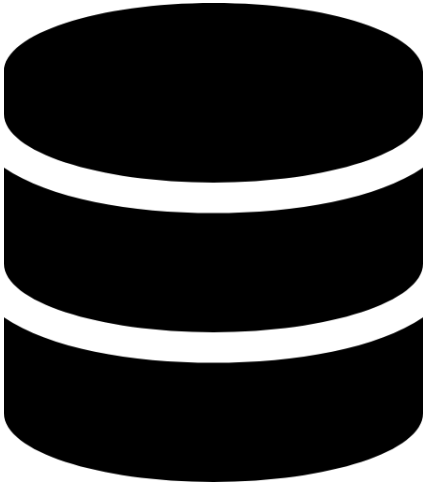
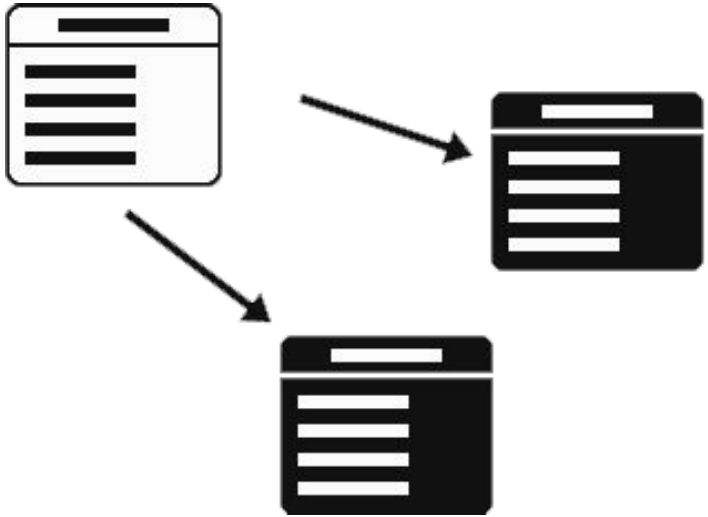
DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas

Abdullah Alsharif,
Gregory M. Kapfhammer, and Phil McMinn

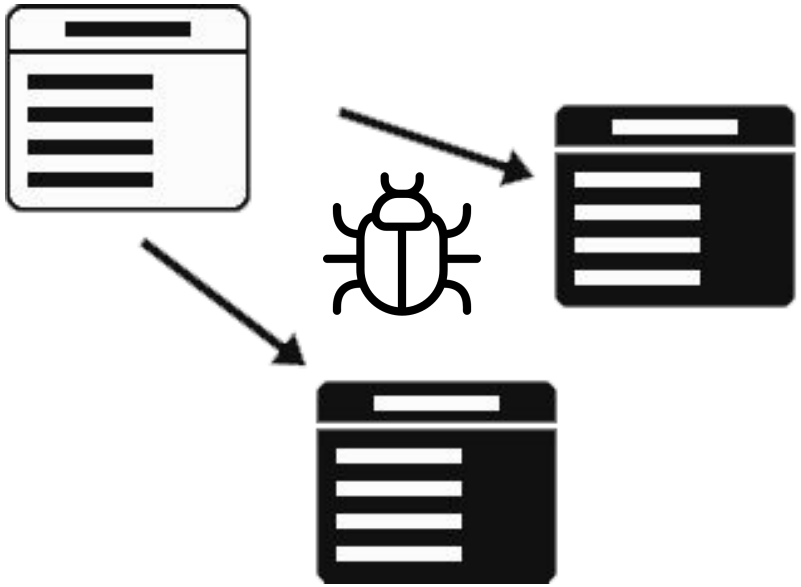


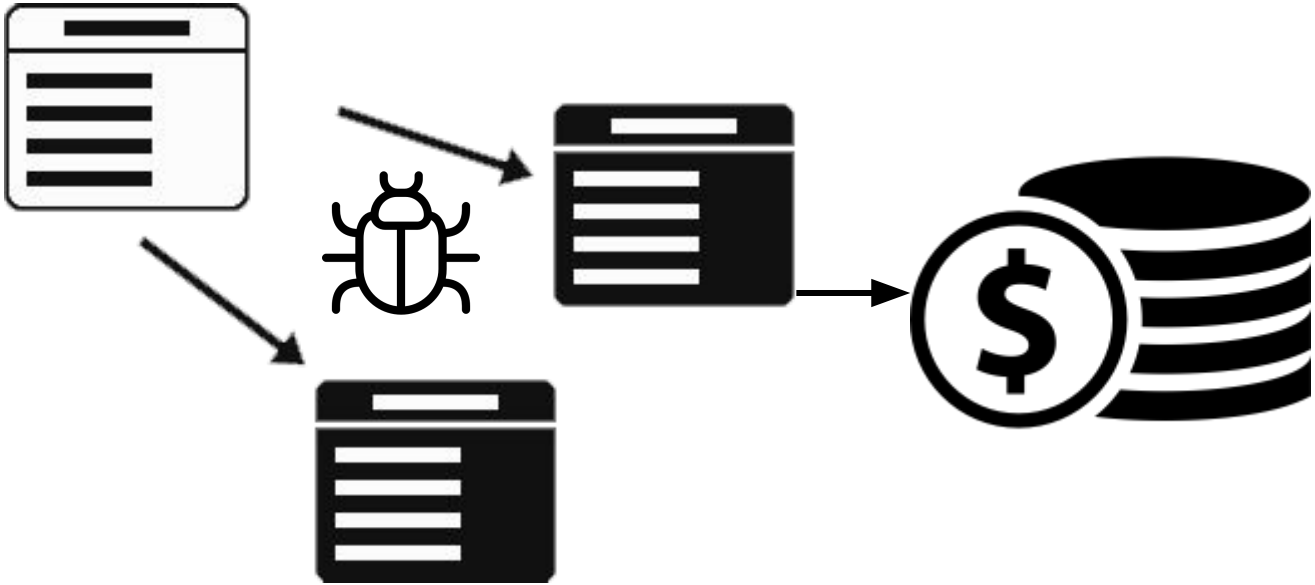


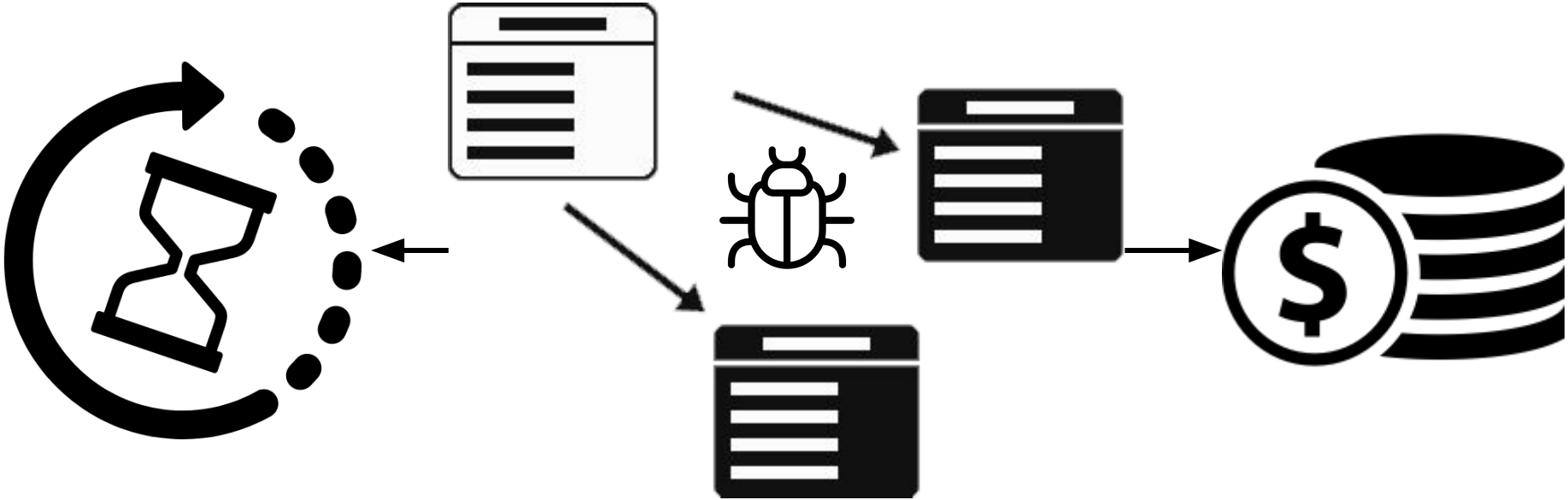
Database

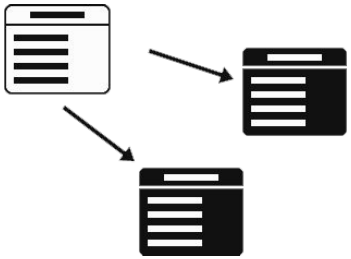


Schema





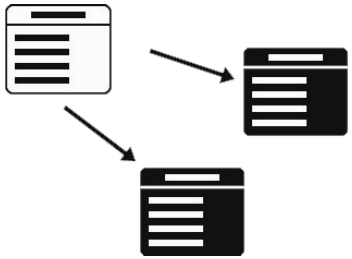




```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

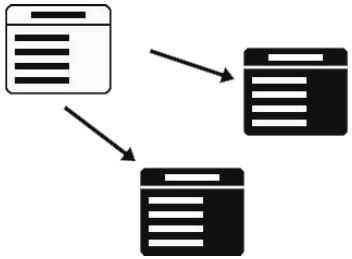
```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```



```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

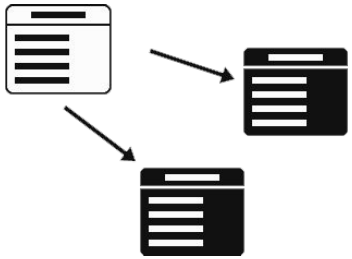
```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```



```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```



```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

Testing Database Schemas Motivation

- Industrial practitioners recommend testing databases (S. Guz, 2011)
- Databases schema, if changed, it need to be tested
- If the DBMS is changed, we need to test schemas behaviour
- Forgetting to add a UNIQUE to a column will duplicate data within a database

Standard

PRIMARY KEY constraint must be NOT NULL

PRIMARY KEY constraint must be NOT NULL

SQLite allows NULLs in a PRIMARY KEY column

PRIMARY KEY constraint must be NOT NULL

SQLite allows NULLs in a PRIMARY KEY column

Follows the standard

PRI ARY KEY straint must be NOT NULL

DEVELOPMENT TO PRODUCTION DEPLOYMENT ISSUES!

KEY column

Follows the standard

```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```

Manual Testing

1)
```
INSERT INTO products(product_no, name, price, discounted_price)
VALUES (0, 'ijyv', 638, 168)
```
✔

2)
```
INSERT INTO orders(order_id, shipping_address)
VALUES (192, 'mrus')
```
✔

3)
```
INSERT INTO order_items(product_no, order_id, quantity)
VALUES (0, 192, 750)
```
✔

4)
```
INSERT INTO products(product_no, name, price, discounted_price)
VALUES (-602, 'ehm', 960, 126)
```
✔

5)
```
INSERT INTO orders(order_id, shipping_address)
VALUES (0, 'u')
```
✔

6)
```
INSERT INTO order_items(product_no, order_id, quantity)
VALUES (0, 192, 64)
```
✘

Manual Testing

1) ```
INSERT INTO products(product_no, name, price, discounted_price)
VALUES (0, 'ijyv', 638, 168)
```
✔

2) ```
INSERT INTO orders(order_id, shipping_address)
VALUES (192, 'mrus')
```
✔

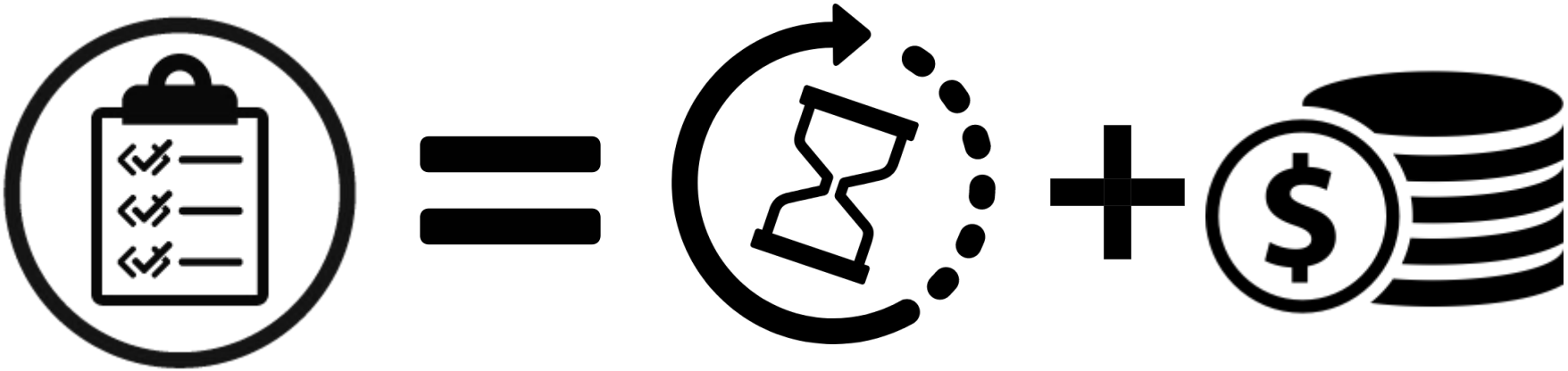
3) ```
INSERT INTO order_items(product_no, order_id, quantity)
VALUES (0, 192, 750)
```
✔

4) ```
INSERT INTO products(product_no, name, price, discounted_price)
VALUES (-602, 'ehm', 960, 126)
```
✔

5) ```
INSERT INTO orders(order_id, shipping_address)
VALUES (0, 'u')
```
✔

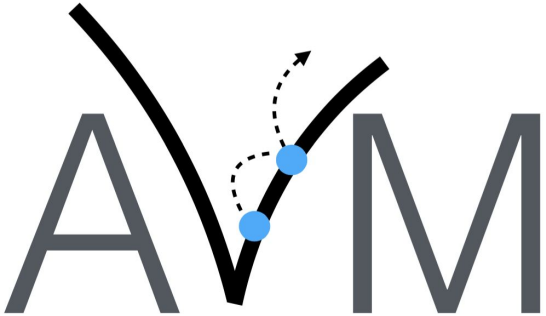
6) ```
INSERT INTO order_items(product_no, order_id, quantity)
VALUES (-602, 192, 64)
```
✔

Manual Database Schema Testing is Challenging



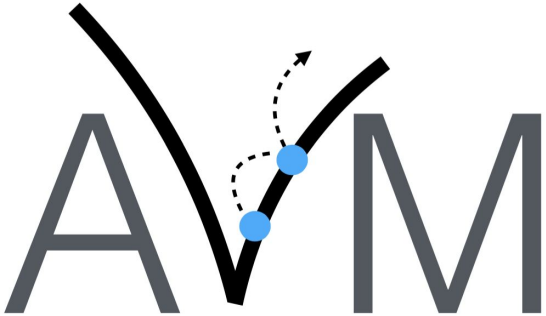
Automated Test Data Generation - Background

- **SchamaAnalyst** is a framework that generates test data for database schemas.
- It has two data generators:
  - **Random+** that uses a pool of constants.
  - The state of the art generator uses **Alternating Variable Method** (AVM).
- It searches for a value for each column involved in the INSERT statement.



Alternating Variable Method - Background

- There are two variants of AVM:
  - **AVM-Random** which uses random values as a starting point for the first generation.
  - **AVM-Defaults** which uses default values (i.e., empty strings for string and 0s for numerics) as a starting point for the first generation. This helps optimise test generation timing.
- The search is evaluated depending on the test requirement, which what drives the search (i.e., fitness function).



Algorithms

```
while ¬termination_criterion
    RANDOMIZE(v⃗)
```

Random+

```
while ¬termination_criterion
    RANDOMIZE(v⃗)
    i ← 1; c ← 0
    while c < n ∧ ¬termination_criterion
        v⃗′ ← MAKEMOVES(v_i)
        if FITNESS(v⃗, r′) < FITNESS(v⃗, r)
            v⃗ ← v⃗′; c ← 0
        else
            c ← c + 1
        i ← (i mod n) + 1
```

AVM

```
1)  INSERT INTO products(product_no, name, price, discounted_price)
                VALUES(10, 'abc', 2, 1);
2)  INSERT INTO products(product_no, name, price, discounted_price)
                VALUES(10, NULL, -1, -1);
```

```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, NULL, -1, -1);
```

```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, NULL, -1, -1);
```

10 11

```
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, NULL, -1, -1);
```

10 11 ☑ NULL 'def' ☑

```
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, NULL, -1, -1);
```

10 11 ☑ NULL 'def' ☑ -1 0 1 ☑

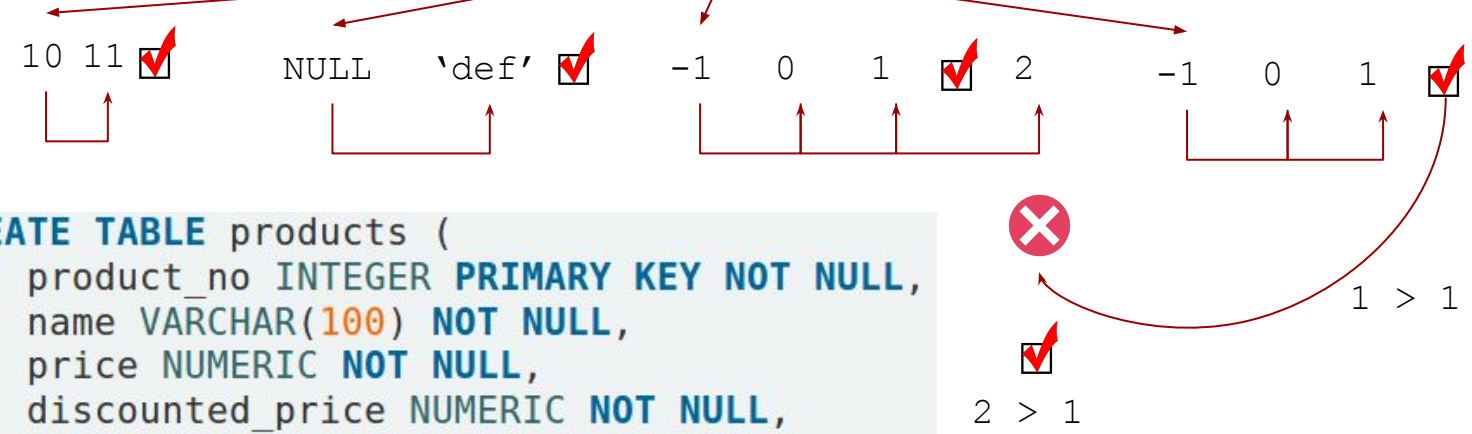
```
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, NULL, -1, -1);
```

10 11 ☑ NULL 'def' ☑ -1 0 1 ☑ -1 0 1 ☑

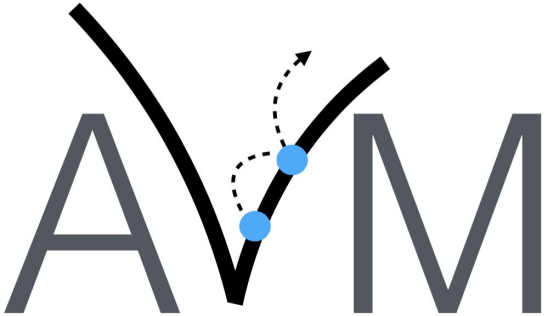
```
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, NULL, -1, -1);
```




```
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Automated Test Generation - Prior Work



Coverage <= 70%

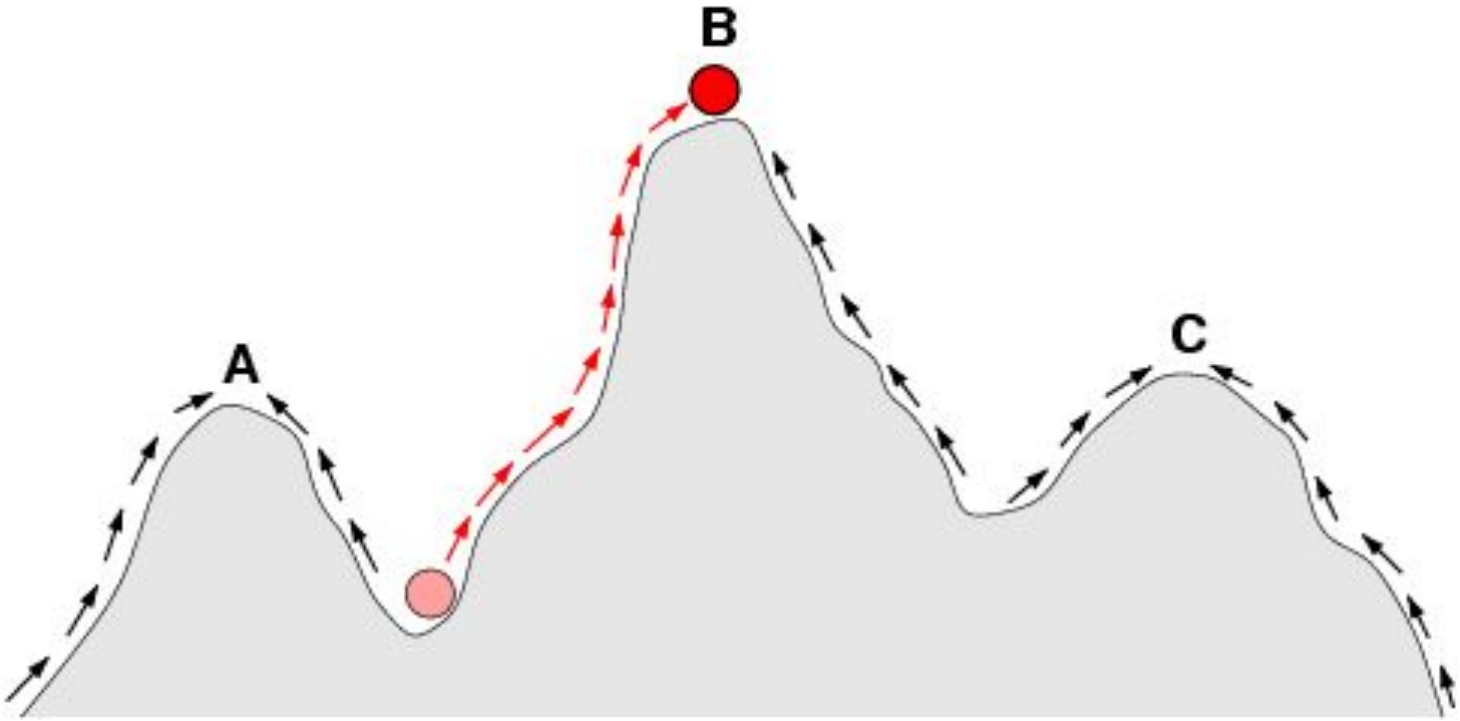
Coverage > 70%

Coverage == 100%

Based on Integrity Constraint Coverage Criterion (McMinn et al, 2015)

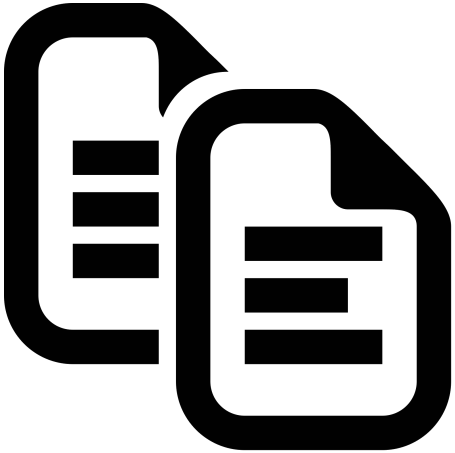
AVM Inefficiencies

AVM Inefficiencies



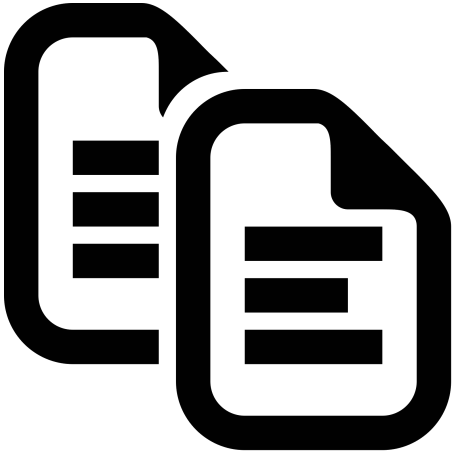
Can we improve ?

Domain Specific Operators



Copying Values

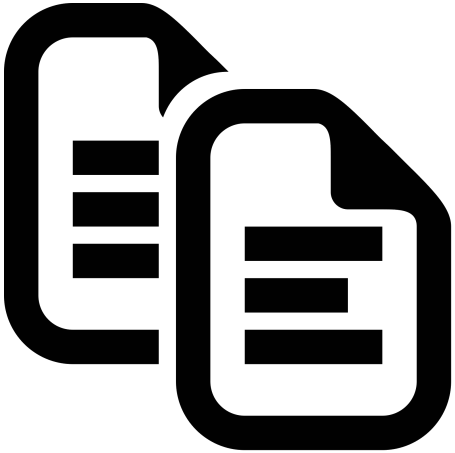
Domain Specific Operators



Copying Values

Flipping NULLs

Domain Specific Operators



Copying Values

Flipping NULLs

Randomise

DOMINO stands for **DOM**ain-specific approach to **IN**tegrity c**O**nstraint test data generation

Algorithms

```
while ¬termination_criterion
    RANDOMIZE(v⃗)
    i ← 1; c ← 0
    while c < n ∧ ¬termination_criterion
        v⃗′ ← MAKEMOVES(v_i)
        if FITNESS(v⃗, r′) < FITNESS(v⃗, r)
            v⃗ ← v⃗′; c ← 0
        else
            c ← c + 1
        i ← (i mod n) + 1
```

AVM

```
RANDOMIZE(v⃗)
while ¬termination_criterion
    COPYMATCHES(v⃗, r)
    RANDOMIZENONMATCHES(v⃗, r)
    SETORREMOVENULLS(v⃗, r)
    SOLVECHECKCONSTRAINTS(v⃗, r)
```

DOMINO

It is like playing DOMINO

Jessica Peterson/Getty Images

```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```

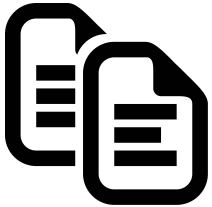
```sql
CREATE TABLE products (
    product_no INTEGER PRIMARY KEY NOT NULL,
    name VARCHAR(100) NOT NULL,
    price NUMERIC NOT NULL,
    discounted_price NUMERIC NOT NULL,
    CHECK (price > 0),
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    shipping_address VARCHAR(100)
);
```

```sql
CREATE TABLE order_items (
    product_no INTEGER REFERENCES products,
    order_id INTEGER REFERENCES orders,
    quantity INTEGER NOT NULL,
    PRIMARY KEY (product_no, order_id),
    CHECK (quantity > 0)
);
```

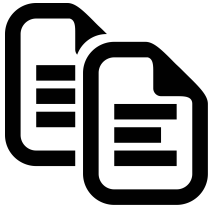
```
INSERT INTO products(product_no, name, price, discounted_price)
                     VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES(-300, 80, 2);
```

```
INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES(-300, 80, 2);
```



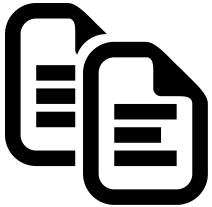
Copying Values

```
INSERT INTO products(product_no, name, price, discounted_price)
                     VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES(-300, 80, 2);
```



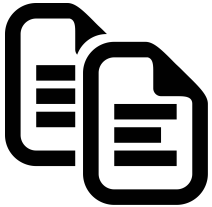
Copying Values

```
INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES(10, 80, 2);
```



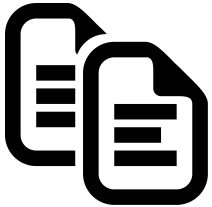
Copying Values

```
INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES( 10, 80, 2);
```



Copying Values

```
INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES( 10, 100, 2);
```



Copying Values

```
INSERT INTO products(product_no, name, price, discounted_price)
                    VALUES(10, 'abc', 2, 1);
INSERT INTO orders(order_id, shipping_address) VALUES(100, 'uvw');
INSERT INTO order_items(product_no, order_id, quantity) VALUES( 10, 100, 2);
```

Flipping NULLs

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, ‘def’, 2, 1);
```

Flipping NULLs

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, ‘def’, 2, 1);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, NULL, 2, 1);
```

Randomise

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, ‘def’, 2, 1);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(28, NULL, 2, 1);
```

Randomise

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, ‘def’, 2, 1);
```

```
1)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(10, 'abc', 2, 1);
2)   INSERT INTO products(product_no, name, price, discounted_price)
                 VALUES(28, ‘def’, 2, 1);
```

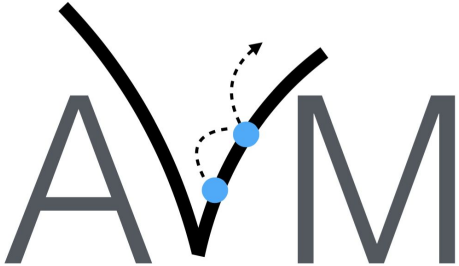
Can we get the best of two worlds?

+

AVM

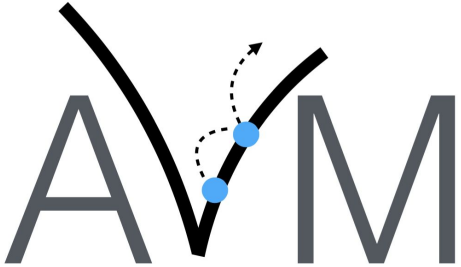
Hybrid Technique

- DOMINO still uses the pool of constants and random picking to solve CHECK constraints.
- AVM is a guided search technique that can help solve CHECK constraints more efficiently.



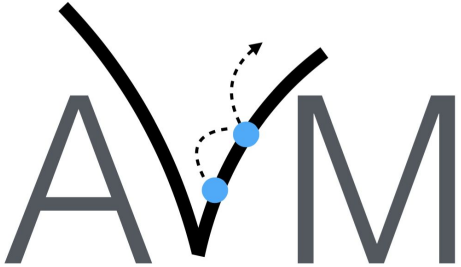
Research Question 1 - Effectiveness and Efficiency

vs

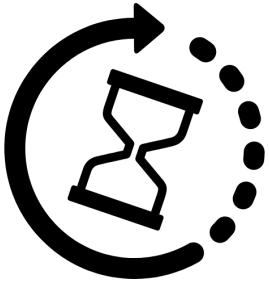


Research Question 2 - Fault-Finding Effectiveness

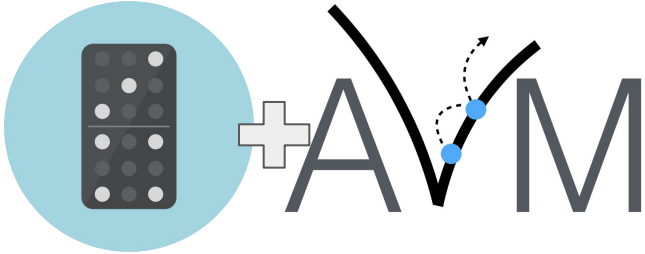
vs



Research Question 3 - DOMINO-AVM Technique

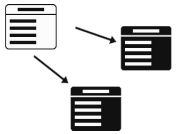


vs

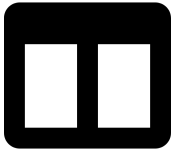


Experimental Setup

Experimental Setup



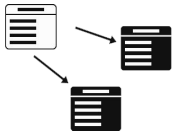
34 Schemas



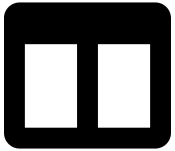
1 to 42 tables
3 to 309 columns

590 ICs

Experimental Setup

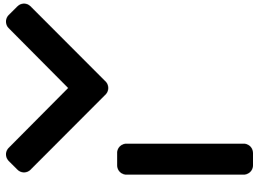


34 Schemas

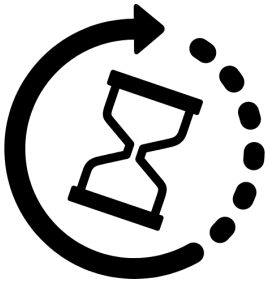


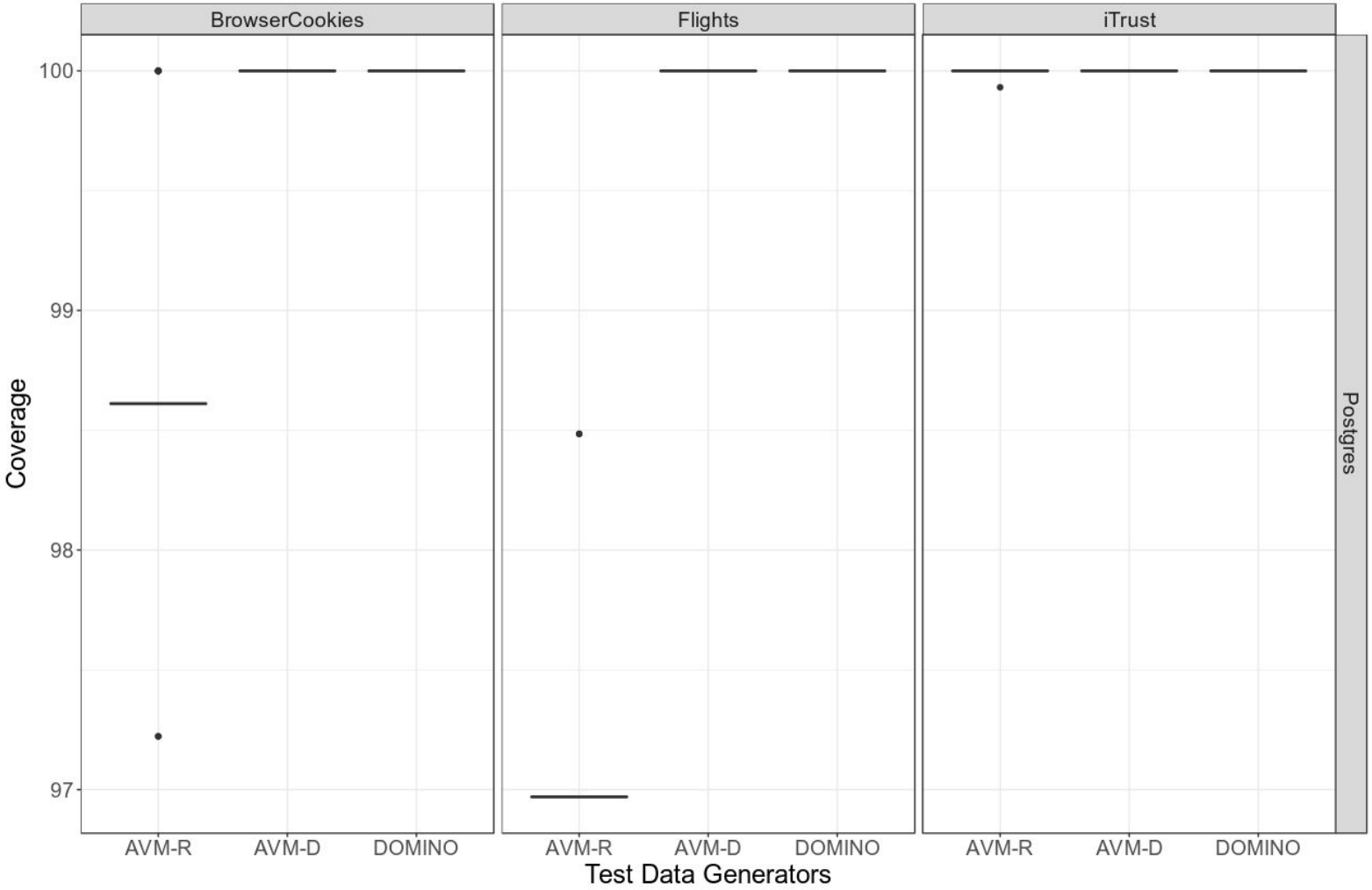
1 to 42 tables
3 to 309 columns

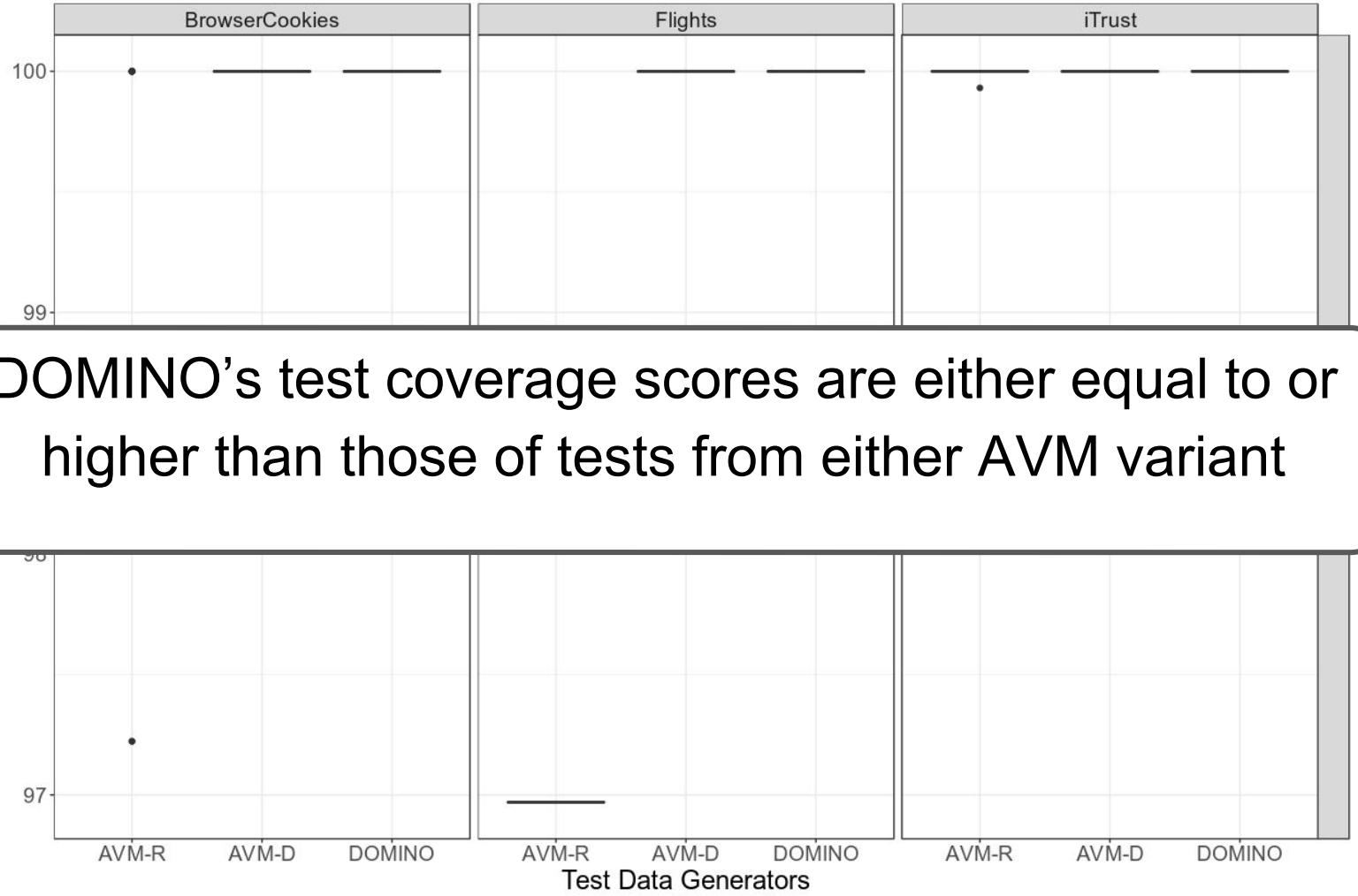
590 ICs



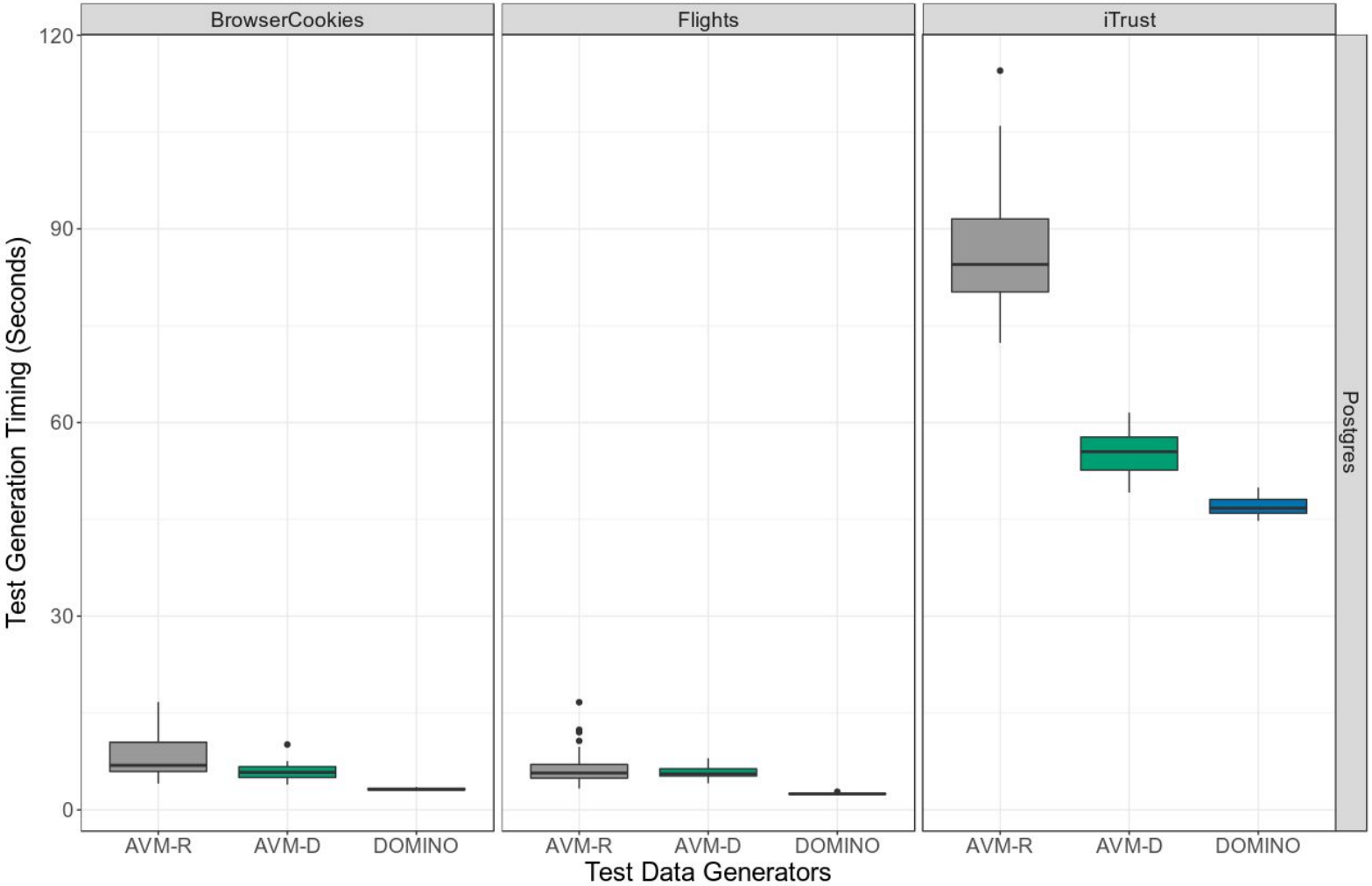
30 Runs

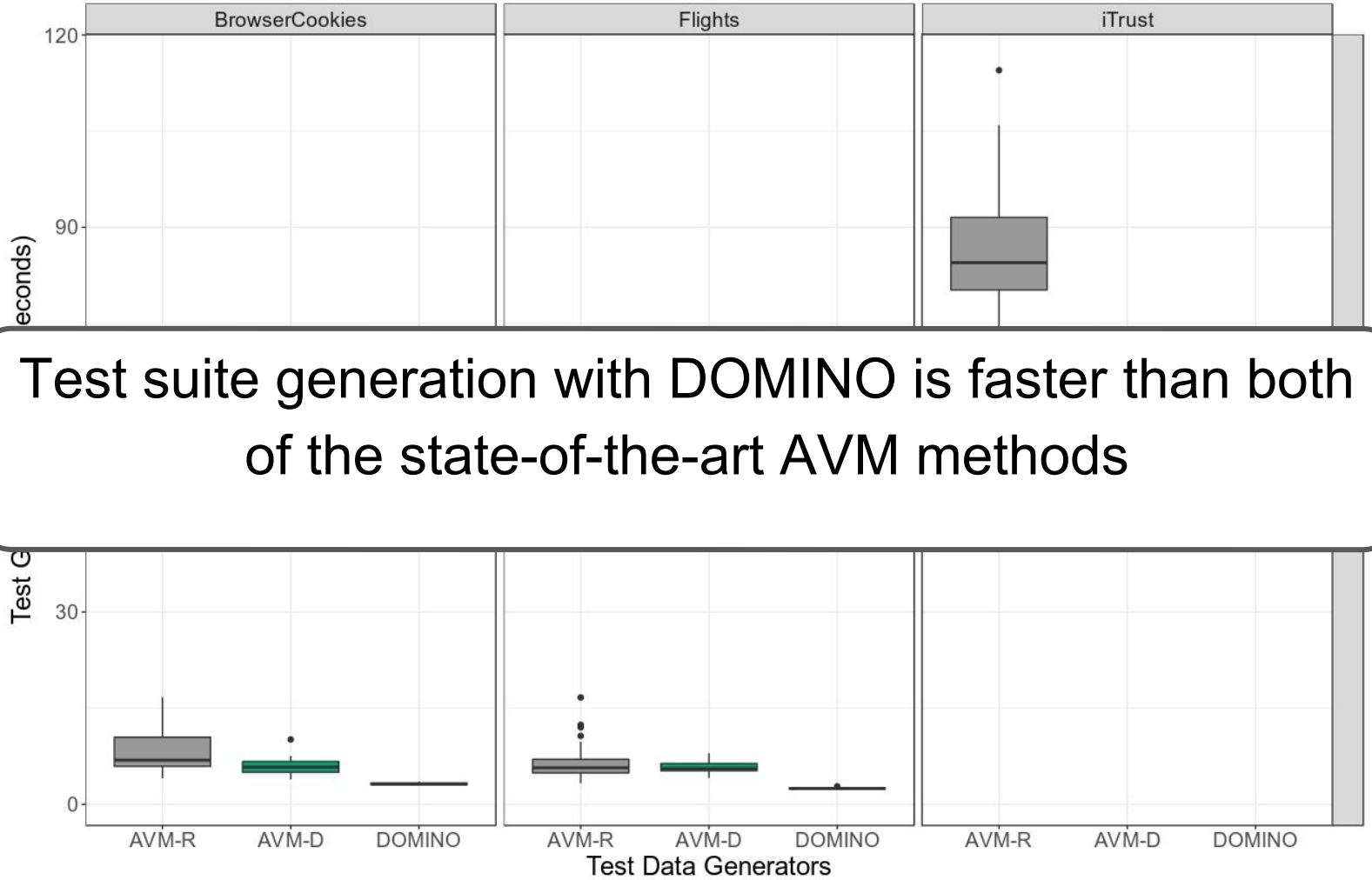



BrowserCookies
Flights
iTrust
Postgres
Coverage
100
99
98
97
AVM-R
AVM-D
DOMINO
Test Data Generators



DOMINO's test coverage scores are either equal to or higher than those of tests from either AVM variant

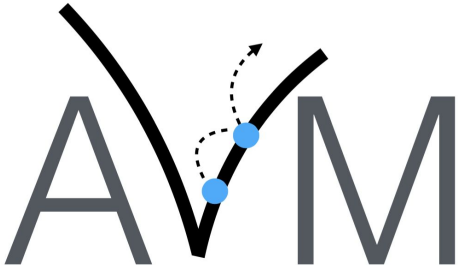

BrowserCookies
Flights
iTrust
Postgres
Test Generation Timing (Seconds)
120
90
60
30
0
AVM-R
AVM-D
DOMINO
Test Data Generators

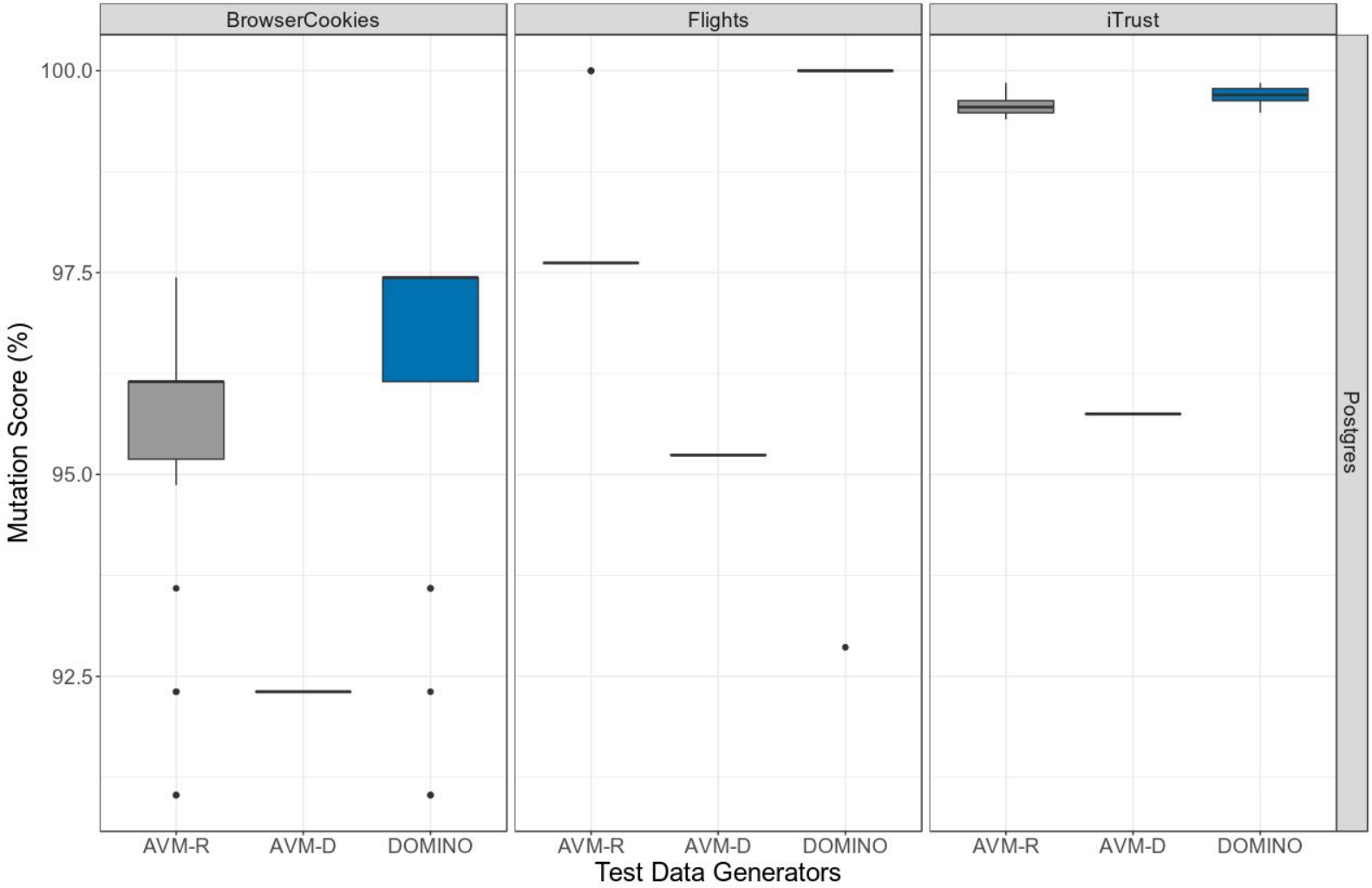


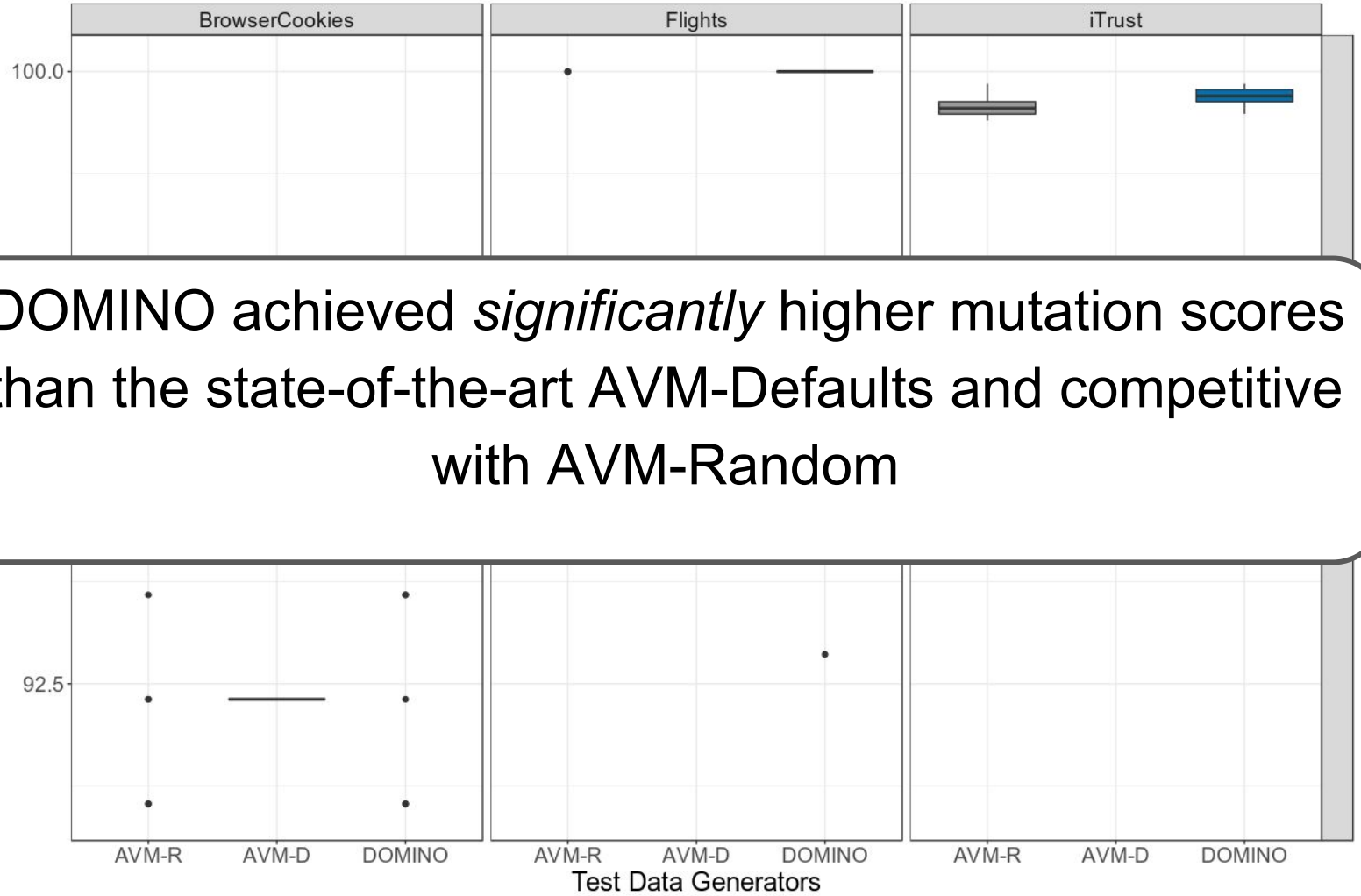
Test suite generation with DOMINO is faster than both of the state-of-the-art AVM methods

Answering RQ1

>



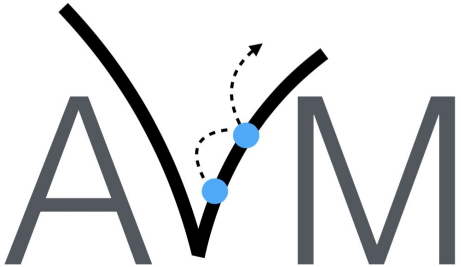

BrowserCookies
Flights
iTrust
Postgres
Mutation Score (%)
100.0
97.5
95.0
92.5
AVM-R
AVM-D
DOMINO
Test Data Generators

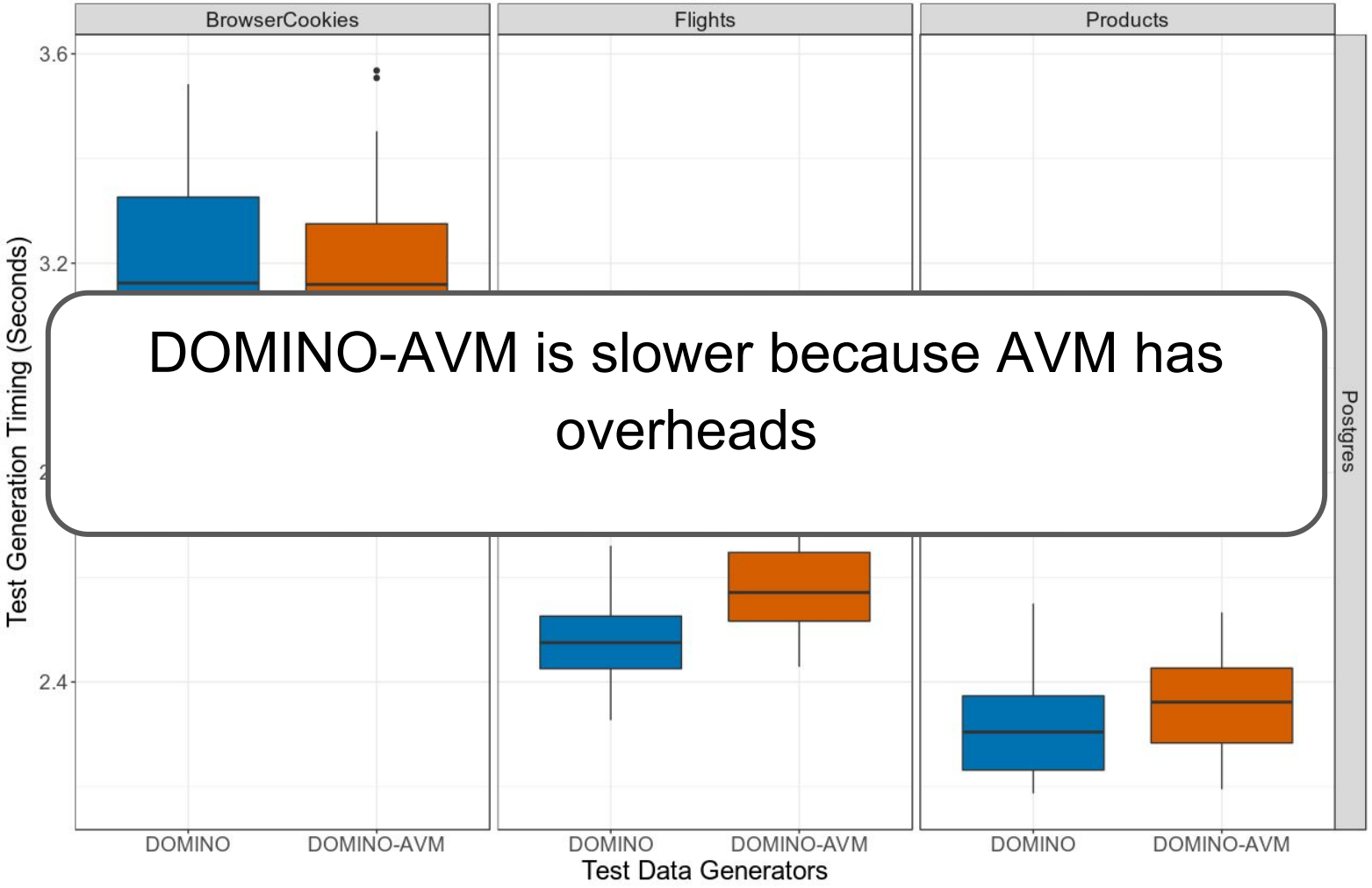


DOMINO achieved *significantly* higher mutation scores than the state-of-the-art AVM-Defaults and competitive with AVM-Random

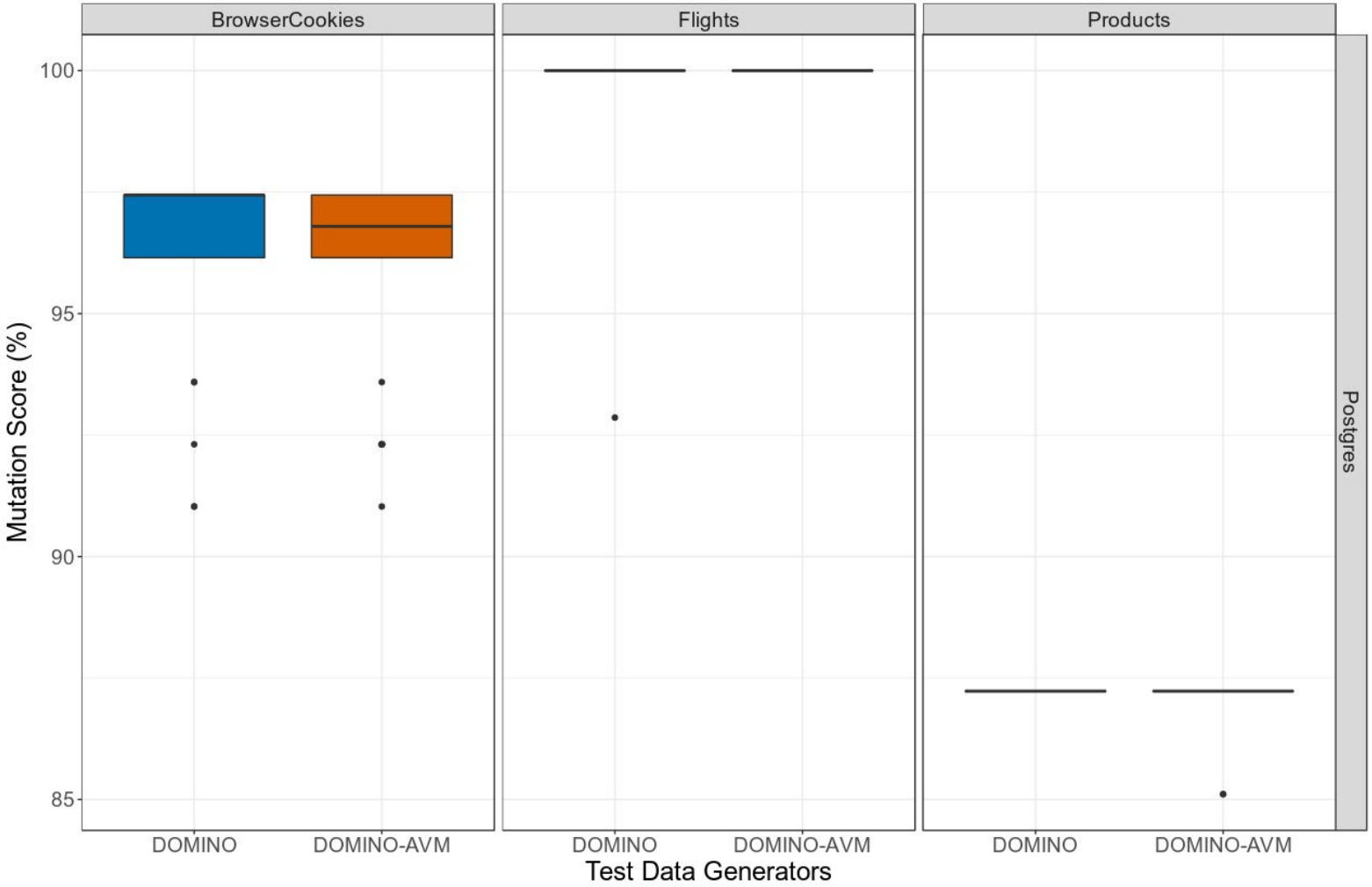
Answering RQ2

>





DOMINO-AVM is slower because AVM has overheads


BrowserCookies
Flights
Products
Postgres
Mutation Score (%)
100
95
90
85
DOMINO
DOMINO-AVM
DOMINO
DOMINO-AVM
DOMINO
DOMINO-AVM
Test Data Generators

Results - Hybrid (DOMINO-AVM)

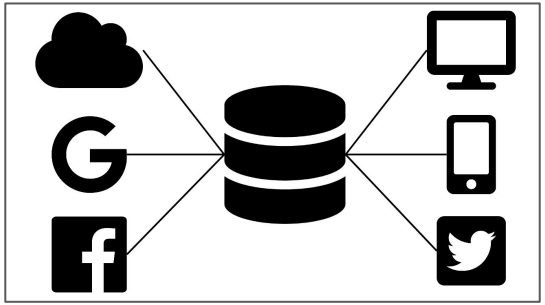
- DOMINO-AVM is significantly better in regard of fault finding than DOMINO for two DBMSs (PostgreSQL & HyperSQL) in just a few cases
  - DOMINO-AVM will generate more diverse data for CHECK constraints
  - DOMINO uses pool of constants which less diverse than guided search
  - We found that constants impact relational operators within CHECK constraints

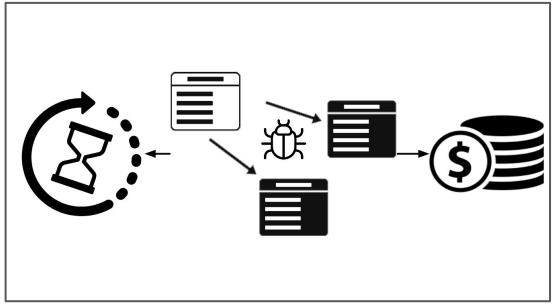
Answering RQ3

- Using AVM has a potential to improve the generation of data involving CHECK constraints is only of benefit for a few cases
- However, the use of random search, as employed by DOMINO, achieves similar results to DOMINO-AVM in a shorter amount of time

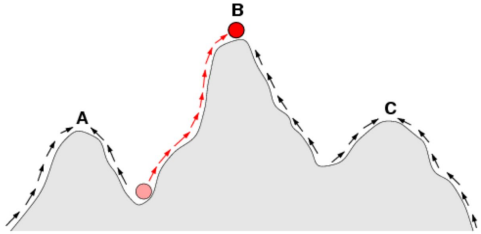
Conclusion and Future Work

- This paper introduced DOMINO, a method for automatically generating test data that systematically exercise the integrity constraints in relational database schemas
- DOMINO uses domain-specific operators and it is extremely competitive and faster to the state-of-the-art methods
- In future, we will look at adding readability to the diverse generated data to help with maintainability
- We are planning to compare DOMINO with more techniques (e.g., Evolutionary Algorithms or/and constraint solvers)





AVM Inefficiencies



Domain Specific Operators

DOMINO stands for **DOM**ain-specific approach to **IN**tegrity c**O**nstraint test data generation

Can we get the best of two worlds?

Experimental Setup