Distancia SQL avanzada para generación de casos de test con EvoMaster

Tesis de Licenciatura en Ciencias de la Computación

Departamento de Computación, Facultad de Ciencias Exactas y Naturales (FCEyN), Universidad de Buenos Aires (UBA)

Alumno: Luis Francisco Brassara
Director: Dr. Juan Pablo Galeotti

Agradecimientos

A Juan Pablo Galeotti por dirigir esta tesis

A Sergio D'Arrigo y Javier Altauz por haber participado del jurado

A mis compañeros por haber hecho la cursada mucho más amena

A mi hermano por mostrarme cómo escribir los IF/FOR en QBasic y así iniciar este camino

Abstract

EvoMaster [7] es una herramienta de generación automática de casos de test para una API REST. Para que la suite generada consiga el mayor coverage posible, EvoMaster intenta generar tests que visiten todas las ramas de los condicionales. Esto lo consigue usando algoritmos evolutivos, cuya población de individuos serán test con llamadas REST y cuya función de fitness será lo que se conoce como branch distance [13]. Como algunas guardas de los condicionales no dependen solo de los parámetros REST sino también del estado de la base de datos, se extendió EvoMaster para tener en cuenta este comportamiento [1]. La heurística planteada en ese trabajo supone que algunas ramas de los condicionales no se visitan cuando la base de datos devuelve una query sin resultados. Luego, identifica esas queries y determina cuán lejos o cerca están de devolver algún resultado mediante la definición de una *SQL distance*. Esa definición de distancia es usada en la fitness function para guiar al algoritmo evolutivo hacia soluciones que visiten más ramas y en consecuencia ofrezcan mayor coverage. Actualmente, la implementación de la SQL distance cubre sólo un subconjunto de todas las posibles queries SQL ya que no tiene soporte para operadores que involucran subqueries tales como EXISTS, IN, ALL/ANY o UNION. Además, la SQL distance actual no da ningún gradiente intermedio cuando la cláusula FROM no trae resultados. El objetivo de la tesis será entonces definir, implementar y probar una extensión de la SQL distancia actual, que llamaremos distancia SQL avanzada, de manera que cubra esos nuevos casos y funcionalidades.

Palabras clave: EvoMaster, generación de tests automáticos, REST APIs, branch coverage, algoritmos evolutivos, algoritmo MIO, branch distance, SQL distance

EvoMaster [7] is an automatic test case generation tool for REST APIs. In order to achieve the highest possible coverage, EvoMaster tries to generate tests that visit all conditional branches. In order to achieve this, it uses evolutionary algorithms, where each individual in the population will be a sequence of REST calls and which fitness function will be what is known as branch distance [13]. Since some conditional guards do not depend only on REST parameters but also on the state of the database, EvoMaster was extended to take this behavior into account [1]. The heuristic proposed in that work assumes that some branches of the conditionals are not visited when the database returns a query without results. Then, it identifies those queries and determines how far or close they are to returning any result by defining a SQL distance. That distance definition is used in the fitness function to guide the evolutionary algorithm towards solutions that visit more branches and consequently offer greater coverage. Currently, the SQL distance implementation covers only a subset of all possible SQL queries as it does not support operators involving subqueries such as EXISTS, IN, ALL/ANY or UNION. Furthermore, the current SQL distance does not give any intermediate gradient when the FROM clause does not bring results. The goal of the thesis will then be to define, implement and test an extension of the current SQL distance, which we will call advanced SQL distance, so that it covers these new cases and functionalities.

Keywords: EvoMaster, automatic test generation, REST APIs, branch coverage, evolutionary algorithms, MIO algorithm, branch distance, SQL distance

Índice general

Marco teórico

Testing

El proceso de testing de un software consiste en validar si dicho software satisface su especificación. Se pueden establecer varias taxonomías de testing según en qué nivel del software se pone el foco (unidad vs. integración vs. sistema), según el grado de conocimiento de la implementación interna (white-box vs black-box), según si los casos de test son creados por humanos o automáticamente, etc. En esta tesis nos interesaremos en particular por tests de tipo white box que ejercitan todo el sistema y son generados automáticamente.

Automated test case generation

Cada caso de test ejecuta el software con una combinación distinta de inputs a fin de poder hallar bugs que comprometan su funcionamiento. En general, la generación automática de casos de test produce una exploración más exhaustiva del espacio de inputs. Además, es considerablemente más rápido generar los tests automáticamente que escribirlos manualmente, por lo cual permite hallar bugs antes en el proceso de desarrollo.

Code coverage

Considerando que un bug es la manifestación de un error en el código, si los casos de test recorren más código entonces es esperable que se hallen más bugs. El *code coverage* es una métrica que representa qué porcentaje del código ha sido visitado durante los tests. Es habitual que se tome como unidad de medida las líneas de código visitadas, en ese caso hablamos de *statement coverage*. También es posible medir qué porcentaje de las ramas o *branches* de los condicionales (if, if else, else if, switch, etc.) fueron visitadas, lo cual se conoce como *branch coverage* o *decision coverage*.

“No es un bug, es un feature”

¿Cómo se determina en los tests generados automáticamente cuando estamos en presencia de un bug? Está claro que si se obtiene una excepción como un null pointer estamos en presencia de un bug, pero no todos los bugs resultan en excepciones. Llamamos *oráculo* a una función que es capaz de decirnos si, dado cierto input, el output es correcto. Una de las mayores dificultades del testing automático es conseguir un oráculo certero.

Search based testing

La generación automática de tests se puede hacer en forma random, pero eso suele llevar a ejercitar solo una parte del código. Para tratar de maximizar el coverage se puede recurrir en cambio a lo que se conoce como *search based testing*. Esta técnica consiste en convertir el problema de hallar casos de test en un problema de optimización combinatoria, el cual es posible de abordar usando metaheurísticas conocidas.

Metaheurísticas

Las metaheurísticas son estrategias para el diseño de algoritmos heurísticos con el fin de resolver problemas de optimización combinatoria. Suelen requerir una representación computacional de las soluciones (por ej. un vector) y una función objetivo o de *fitness* que evalúa que tan buena es una solución. A partir de eso, exploran el espacio de soluciones con estrategias de búsqueda que dependen de la metaheurística.

Algunas metaheurísticas tienden a hacer búsquedas locales (hill climbing, tabu search, etc.) y otras tienden a hacer búsquedas globales (genetic algorithms, particle swarm optimization, etc.). Muchas veces las metaheurísticas cuentan con hiperparámetros que permiten establecer el balance entre exploración local y global (simulated annealing, etc.).

Algoritmos evolutivos

Es una familia de metaheurísticas basadas en imitar la evolución de las especies, utilizando conceptos de ese dominio tales como mutaciones y cruzas de individuos. En términos computacionales, una mutación será producir una nueva solución alterando una solución ya existente (por ej. cambiar una coordenada si la solución es un vector) y una cruza será producir una nueva solución mezclando dos soluciones ya existentes (por ej. devolver un vector que es la concatenación de la mitad de un vector y la mitad del otro). En cualquiera de los dos casos, hay que procurar siempre que el nuevo individuo represente una solución válida.

El más simple de estos algoritmos es el *1+1 evolutionary algorithm*. Comenzando con 1 individuo random, en cada iteración ese individuo es mutado, dando lugar a 1 hijo. Si el hijo es mejor que el padre, lo reemplazará y pasará a ser la mejor solución hallada hasta el momento.

Probablemente el ejemplo más representativo de los algoritmos evolutivos es *genetic algorithms* (GA). En GA ya no tenemos un individuo sino una población de individuos. Comenzando con una población random, se la va haciendo evolucionar del siguiente modo. Primero, se seleccionan los mejores individuos de la población en términos de la función de fitness. A estos individuos se los conoce como *elite*. Luego, se crea una nueva población con la elite (o una parte de ella) y con una nueva camada de individuos surgidos a partir de mutaciones y cruzas entre la elite.

Otros conceptos de la naturaleza que suelen ser usados en algoritmos evolutivos son *cromosoma* y *gen*. Se denomina cromosoma al conjunto de variables de la solución (por ej. si la solución es un par ordenado, el cromosoma serán los campos x e y) y gen a cada una de las variables de la solución (siguiendo el ejemplo anterior, x es será un gen e y otro gen). Notar que, dado que el cromosoma determina la estructura de la solución, su elección condicionará las operaciones de mutación y cruza.

REST

Se conoce como REST (representational state transfer) a un estilo de arquitectura para web services. Está basado en recursos (por ej. un producto en un ecommerce), que son unívocamente identificados por una URI jerárquica (por ej. /products representa todos los productos y /products/{id} representa al producto identificado como {id}) y manipulados mediante métodos HTTP (por ej. GET /products para obtener todos los productos o GET /product/{id} para obtener el producto {id}, POST /products para crear un producto nuevo, DELETE /product/{id} para borralo, etc.).

EvoMaster

Overview

EvoMaster (www.evomaster.org) es una herramienta de generación automática de casos de test para aplicaciones web. Para la generación de tests utiliza algoritmos evolutivos, de ahí el nombre de la herramienta.

El testing que realiza Evomaster es a nivel de sistema ya que prueba los web services de la aplicación. Fue desarrollado mayormente para APIs REST, pero también soporta otras arquitecturas (GraphQL, gRPC, etc.). En caso de testear una API REST es necesario proveer la especificación de la API usando OpenAPI/Swagger.

EvoMaster es fundamentalmente una herramienta de white-box testing. Si bien es posible hacer black-box testing, los resultados pueden no ser los óptimos. Para poder llevar a cabo el white-box testing, EvoMaster instrumenta el código de la aplicación a testear y de esta manera es capaz de identificar qué porciones del código han sido visitadas por un caso de test.

Para la detección de bugs, EvoMaster utiliza como oráculo el status HTTP que devuelven los webservices. Según la especificación de HTTP, los status 2xx corresponden a una llamada exitosa, los 3xx a redirecciones, los 4xx a un error del

cliente y los 5xx a un error en el servidor, por lo que el oráculo concluye que hay un bug cuando el status es 5xx.

Dentro de los algoritmos evolutivos que puede utilizar EvoMaster se encuentran Whole Test Suite (WTS), Many-Objective Sorting Algorithm (MOSA) y Many Independent Objective (MIO). Si bien no hay uno que sea el mejor en todos los casos, el algoritmo MIO es el que tiene mejor performance en general.

Algoritmo Many Independent Objective (MIO)

MIO es un algoritmo evolutivo específico para el problema de hallar casos de test. Este problema tiene varias particularidades. Una de ellas es que maximizar el coverage implica satisfacer no uno sino múltiples objetivos (*targets*). Como ya se mencionó, el coverage puede ser medido por ejemplo a partir de los branches visitados. En ese caso, cada branch será un objetivo a satisfacer. Otra particularidad de este problema es que los objetivos pueden ser satisfechos independientemente uno del otro porque la solución, un conjunto de casos de tests (*test suite*), puede ser incremental. Esto significa que para cubrir los objetivos restantes se pueden ir agregando tests a los ya existentes, aunque claramente se debe tratar de que el test suite sea minimal.

El pseudocódigo (tomado de [8]) es el siguiente:

Algorithm 1: Pseudo-code of the MIO Algorithm [9]

Input : Stopping condition C, Fitness function δ, Population size n, Probability for random sampling P_r, Start of focused search F
Output : Archive of optimised individuals A

```
1  T ← SetOfEmptyPopulations()
2  A ← {}
3  while ¬C do
4      if P_r > rand() then
5          p ← RandomIndividual()
6      else
7          p ← SampleIndividual(T)
8          p ← Mutate(p)
9      end
10     foreach element k ∈ ReachedTargets(p) do
11         if NewTarget(k) then
12             T ← T ∪ T_k
13         end
14         T_k ← T_k ∪ {p}
15         if IsTargetCovered(k) then
16             UpdateArchive(A, p)
17             T ← T \ T_k
18         else if |T_k| > n then
19             RemoveWorst(T_k, δ)
20         end
21     end
22     UpdateParameters(F, P_r, n)
23 end
```

En líneas generales, el algoritmo MIO mantiene una población de tests para cada objetivo (por ej. un branch determinado en el código) formada por aquellos tests que más cerca estuvieron de satisfacerlo. Al comenzar, la población para cada objetivo está vacía. Luego, en cada iteración, crea un nuevo test al azar o mutando uno existente de alguna población. Al ser ejecutado, ese test alcanzará algunos objetivos, algunos de los cuales cubrirá y otros no (por ej. un test puede alcanzar un branch pero no cubrirlo si la condición del branch no se cumple). Para cada objetivo cubierto, el test pasa a ser parte de la solución para ese objetivo (si ya había otro test para ese objetivo, se queda con el más corto) y se descartan todos los tests anteriores de esa población. En cambio, para cada objetivo no cubierto, se agrega el test a la población de ese objetivo. La población de cada objetivo tiene un tamaño máximo y en caso de ser excedido se elimina el test que más lejos está de cubrir el objetivo. Dicha distancia entre un test y el objetivo es determinada por la función de fitness (por ej. branch distance si los objetivos son branches).

Notar que en la última instrucción se actualizan los parámetros, entre ellos la probabilidad con la que se crea un nuevo test al azar (en vez de mutar uno existente). A través de las sucesivas iteraciones, esta probabilidad irá disminuyendo, haciendo que MIO pase de una búsqueda global a enfocarse en lo local.

Testing de APIs REST

MIO es un algoritmo específico para hallar casos de test, no obstante, para cada dominio en particular hará falta definir cómo se representarán los tests y cómo se harán las mutaciones (no hay cruzas en MIO). Además, también habrá que definir cuáles serán los objetivos a satisfacer y cuál será la función de fitness que determine la distancia a esos objetivos.

Para el testing de APIs REST, Evomaster representa al test como una secuencia de llamadas HTTP. Las posibles llamadas HTTP que soporta la API son tomadas de la especificación de OpenAPI y la secuencia en general suele seguir patrones comunes de REST, como por ejemplo crear un recurso con POST y después tratar de obtenerlo mediante GET.

Entonces, en términos de programación evolutiva, el cromosoma será una secuencia de llamadas HTTP. Por su lado, los genes serán cada una de las partes variables de esas llamadas como los query y los path parameters de la URI, los campos en el JSON del body, etc. Sobre ese cromosoma y esos genes, EvoMaster realiza dos tipos de mutaciones, una mutación estructural donde agrega o saca llamadas HTTP de la secuencia y otra mutación donde mantiene la estructura, pero modifica el valor de uno o más genes.

Por último, queda por definir los objetivos y la función de fitness para esos objetivos. EvoMaster establece varios tipos de objetivos como pueden ser la cobertura de statements y branches y la obtención de faults (que un endpoint devuelva un status 5xx). En la presente tesis, nos interesarán particularmente los objetivos de cobertura de branches, en los cuales se necesitará una función de fitness que pueda establecer qué tan cerca está un test de cubrir un branch. Para eso, usaremos la *branch distance*.

Branch distance

La *branch distance* es una distancia popular en la literatura de search based testing que permite cuantificar qué tan cerca están las variables presentes en la condición de un branch de hacer que dicha condición se cumpla. Supongamos que tenemos el siguiente branch con la variable a:

```
if(a == 1){
  //…
}
```

Está claro que a = 2 está más cerca de cumplir la condición que a = 100 y en consecuencia desearíamos que la distancia sea menor en el primer caso. Veamos si esto sucede computando la branch distance según la siguiente tabla:

SET I.

Predicate θ	Function $\delta_\theta(I)$
A	if a is TRUE then 0 else k
$A = B$	if $abs(a-b) = 0$ then 0 else $abs(a-b)+k$
$A \neq B$	if $abs(a-b) \neq 0$ then 0 else k
$A < B$	if $a-b<0$ then 0 else $(a-b)+k$
$A \leq B$	if $a-b \leq 0$ then 0 else $(a-b)+k$
$A > B$	$\delta_{B<A}(I)$
$A \geq B$	$\delta_{B\leq A}(I)$
$\neg A$	Negation is moved inward and propagated over A
$A \wedge B$	$\delta_A(I)+\delta_B(I)$
$A \vee B$	$min(\delta_A(I),\delta_B(I))$

Efectivamente, en el caso anterior la branch distance para a = 2 será de 1 mientras que para a = 100 será de 99 (obviamos la constante por simplicidad). Veamos otro ejemplo más complejo:

```
if(a != 1 && b <= 0){
  //…
}
```

¿Cuál es la branch distance para a = 2 y b = 100?

Según la tabla, cuando hay un operador de conjunción las distancias se suman, lo cual es razonable ya que ambas condiciones se tienen que cumplir al mismo tiempo. La distancia a la condición a != 1 es 0 (a = 2) y la distancia a b <= 0 es 100 (b = 100). Sumando ambas llegamos a que la branch distance es 100. Un último ejemplo:

```
if(!(a != 1 && b > 0)){
  //…
}
```

¿Cuál es la branch distance para a = 2 y b = 100?

Cuando hay una negación como en este caso, la definición de branch distance en la tabla demanda que sea propagada hacia adentro. Aplicando DeMorgan, la condición resultante será:

```
if(a == 1 || b <= 0){
  //…
}
```

Ahora, el operador es de disyunción. Según la tabla, hay que tomar el mínimo de las dos distancias, lo cual tiene sentido ya que alcanza con cumplir solo una de las condiciones. La distancia a la condición a == 1 es 1 (a = 2) y la distancia a b <= 0 es 100 (b = 100). Tomando el mínimo, llegamos a que la branch distance es 1.

Por simplicidad, en los ejemplos se omitió la constante k. Esta constante se suma a la distancia para aplicar una penalización en los casos que la condición no se cumple. No debe ser demasiado baja, ya que no habría mucha diferencia en la distancia entre cumplir o no la condición (notar que al menos debe ser mayor que cero para que exista dicha diferencia). Tampoco puede ser excesivamente grande porque sino ese factor dominaría toda la distancia independientemente de que ocurre con la condición. Experimentalmente, el valor de 1 da buenos resultados.

Si bien es cierto que la branch distance se aplica a las variables del branch, que podrían no coincidir con los inputs de un test, también es cierto que en general el

valor de esas variables dependen de los inputs de un test. Luego, aunque de forma indirecta, la branch distance también brinda una distancia para los inputs del test.

Heurística para SQL

Motivación

EvoMaster lleva a cabo testing del tipo white-box generando tests que vayan cubriendo objetivos específicos en el código como los branches. Para saber qué inputs debería usar un test para cubrir o al menos acercarse a un objetivo, se utiliza una función de fitness como la branch distance. Esta función provee un gradiente que actúa como una guía para la elección de esos inputs.

El enfoque anterior se basa fuertemente en que el comportamiento de la API depende solo de los inputs del test. Sin embargo, eso no siempre es cierto, ya que es habitual que las APIs interactúen por ejemplo con una base de datos. Entonces, al correr un test, el path de ejecución podría depender también del estado de la base de datos. Veamos un ejemplo:

```
List<Offer> offers = findOffers();
if(!offers.isEmpty()){
  //...
}

public List<Offer> findOffers(){
   //query the database
}
```

En el caso anterior, la función de fitness que venimos usando no podrá proveer ningún feedback de qué tan cerca o lejos nos encontramos de cubrir el branch ya que su condición no depende de los inputs del test sino del estado de la base de datos. Surge entonces la necesidad de que la función de fitness pueda dar gradiente también en estos casos.

Heurística

En el trabajo “SQL Data Generation to Enhance Search-Based System Testing” (2019), A. Arcuri y J.P. Galeotti idearon una manera de poder alcanzar objetivos que dependen del estado de la base de datos. En particular, se enfocaron en las bases de datos de tipo SQL.

En el contexto de SQL, los objetivos que dependen del estado de la base en concreto dependerán del resultado de las queries **SELECT**. Según el resultado de estas queries, se pueden tomar distintos paths de ejecución que cubren unos u otros objetivos. No obstante, a veces las queries no dan ningún resultado porque la secuencia de llamadas HTTP del test no fue capaz de generar las entradas en la tabla correspondiente o porque hay entradas, pero no se cumple la cláusula **WHERE** para ninguna de ellas.

Para tratar de evitar estas queries con resultados vacíos, se establecerá un nuevo tipo de objetivo. Así como anteriormente se buscaba cubrir statements, branches y faults, ahora también se buscará que las queries devuelvan algún resultado. Este nuevo tipo de objetivo para las queries requerirá también de una nueva definición de distancia. Del mismo modo que usamos la branch distance para la cobertura de branches, para las queries usaremos la SQL *distance*, que será analizada en detalle en la próxima sección.

Mientras que statements, branches y faults serán objetivos primarios, las queries serán objetivos secundarios ya que no son un fin en sí mismo sino un medio para destrabar más branches y así poder cubrir más porciones del código. El modelado de las queries como objetivo también será distinto. No se agregarán queries como objetivos a la par de statements, branches, etc. sino que se modificará la función de fitness para que pondere positivamente aquellos tests con menos queries vacías. De esta manera, si dos tests tienen la misma branch distance, entonces se usará como criterio de desempate la SQL distance (en rigor, un promedio de la SQL distance sobre las queries vacías del test).

Luego, la nueva función de fitness combina branch distance y SQL distance y, a diferencia de su versión anterior, es capaz de dar gradiente también cuando los objetivos dependen del estado de la base de datos.

Continuando con la misma metodología white-box de antes, bajo la cual se instrumentaba el código para calcular la branch distance, ahora además se interceptarán las queries SQL realizadas desde la aplicación a la base de datos para así poder calcular la SQL distance.

Distancia SQL

Resta definir cómo se calcula efectivamente la SQL distance. Supongamos que tenemos la siguiente query:

SELECT salary **FROM** Employees **WHERE** salary >= 2000

Si la query trae algo, la distancia es 0. Si no trae nada, se saca la cláusula **WHERE** y se vuelve a ejecutar la query:

SELECT salary **FROM** Employees

Si sigue sin traer nada, se devuelve la distancia máxima (usando un número muy grande como el máximo Double posible). Si esta vez sí trajo algo, se puede calcular la distancia entre cada una de las filas (*rows*) que trae y la cláusula **WHERE** que tenía la query, quedándonos con la mínima de esas distancias.

La distancia entre una fila y la cláusula **WHERE** se calcula en forma análoga a la branch distance. Supongamos que tenemos la siguiente tabla:

Employees		
ID	Name	Salary
1	John	1500
2	Paul	1900

Por un lado, la distancia entre la primera fila y la condición salary >= 2000 será de 500 (salary = 1500). Por el otro, la distancia entre la segunda fila y la condición será

de 100 (salary = 1900). Quedándonos con la distancia mínima entre ambas filas, la SQL distance resulta ser de 100.

La SQL distance soporta los mismos operadores que la branch distance, pero en su versión SQL (**AND** en vez de **&&, OR** en vez de **||,** etc.).

Formalmente:

SQL distance = **D-Query**(query)

D-Query(SELECT ... FROM <ROW-SET> WHERE <CONDITION>) =
    IF query return results THEN
        0 (zero)
    ELSE
        D-Condition(<CONDITION>, <ROW-SET>)

D-Condition(<CONDITION>, <ROW-SET>) =
    IF <ROW-SET> is empty THEN
        MAX_DISTANCE
    ELSE
        min { δ_r(<CONDITION>) | r ∈ <ROW-SET> }

donde δ_r es la branch distance (ver tabla en sección “Branch distance”)

Transformación de queries

Antes de calcular la SQL distance, las queries pueden necesitar algunas transformaciones. Supongamos ahora que tenemos la siguiente query:

SELECT city **FROM** Employees **WHERE** city = “CABA” AND salary >= 2000

Si removemos la cláusula **WHERE** quedará:

SELECT city **FROM** Employees

Sin embargo, si dejamos la query así no podremos calcular la SQL distance porque las filas obtenidas incluirán el campo city, pero no el campo salary, que también está presente en la condición del **WHERE**. Resulta entonces necesario antes de ejecutar la query reescribirla como:

SELECT city, salary **FROM** Employees

Consideremos ahora el caso de las agregaciones:

SELECT COUNT(*) **FROM** Employees **WHERE** salary >= 2000

Esta query así escrita devolverá un número en vez de un conjunto de filas sobre los que aplicar la SQL distance. En este caso, al quitarle el **WHERE** hay que reescribirla como:

SELECT salary **FROM** Employees

Por último, se removerá cualquier cláusula que pueda llegar a cambiar la cantidad de filas devueltas como **GROUP BY**, **LIMIT**, etc.

Inserciones

A veces las queries dan resultados vacíos porque la secuencia de llamadas HTTP requerida para popular las tablas puede ser demasiado compleja de hallar. Otras veces, directamente no existe esa secuencia porque la base de datos es read only (no es el servicio que estamos probando el que la popula). Es necesario en estos casos realizar inserciones que permitan recorrer aquellos paths de ejecución dependientes del estado de la base de datos y así maximizar el coverage.

Entonces, a la secuencia de llamadas HTTP que ya tenía el test ahora se le agregará una secuencia de inserciones SQL. Estos **INSERT** de SQL se ejecutarán antes que las llamadas, poniendo a la base de datos en un estado determinado antes del test en sí. En términos de algoritmos evolutivos, se amplía el cromosoma con inserciones SQL y también se agregan genes, correspondientes a las columnas de las filas que serán insertadas.

Como siempre, cambios en el cromosoma y los genes implican cambios en los operadores de búsqueda, en este caso las mutaciones. Al igual que con las llamadas HTTP, las mutaciones serán agregar o quitar inserciones o modificar valores de las filas a insertar. Sin embargo, requieren un cuidado especial ya que algunas inserciones podrían violar constraints de las tablas.

Para evitar inserciones que no afectan en nada el coverage y extienden innecesariamente el test, se procura solo agregar inserciones en los tests cuando al correr los mismos se detectan queries vacías. Además, estas inserciones solo serán sobre tablas a las que hacen referencia esas queries.

Distancia SQL avanzada

En este trabajo se definió, desarrolló y testeó una nueva versión de la SQL distance. Llamaremos *distancia avanzada* a esta nueva versión y *distancia estándar* a la versión anterior.

Limitaciones anteriores

La principal limitación de la distancia estándar es que no soporta subqueries. Las subqueries pueden ser:

a) Usadas por operadores específicos para subqueries:

i) **EXISTS**:

```
SELECT * FROM t1 WHERE EXISTS
    (SELECT * FROM t2 WHERE ...)
```

ii) **ALL** y **ANY/SOME**:

```
SELECT * FROM t1 WHERE a = ALL (SELECT * FROM t2)
```

iii) **IN:**

```
SELECT * FROM t1 WHERE a IN (SELECT * FROM t2)
```

b) Usadas en la cláusula **FROM**:

SELECT * FROM (SELECT * FROM t1 **WHERE ...)** t2 **WHERE ...**

o incluso combinado con JOINs:

SELECT * FROM (SELECT * FROM t1 **WHERE ...)** t2 **JOIN** t3 **WHERE ...**

c) Usadas en la cláusula **UNION / UNION ALL**:

(SELECT * **FROM** t1) **UNION (SELECT** * **FROM** t2)

d) Usadas como valor:

SELECT * **FROM** t1 **WHERE** a **= (SELECT** b **FROM** t2 **WHERE** ...**)** - 1

Además, la distancia estándar solo usa la cláusula **WHERE** para dar gradiente. No obstante, también hay información valiosa en la cláusula **FROM**. Esto es especialmente cierto cuando en la misma hay un **JOIN** o una subquery, porque cuanto más cerca estén las condiciones del JOIN de cumplirse o la subquery de dar resultados también se estará más cerca de que la query original de resultados.

En la siguiente sección se definirá la distancia avanzada, la cual soporta subqueries y es capaz de dar gradiente usando también la cláusula **FROM**. Además, en la sección “Features” se encontrarán nuevos operadores y tipos de datos soportados por la distancia avanzada.

Nueva definición

Truthness

Antes que nada, se define una nueva estructura de datos llamada *Truthness*. Es una tupla (ofTrue, ofFalse) donde ofTrue y ofFalse son de tipo Double. Tiene como invariante que ofTrue y ofFalse están en el rango [0, 1] y que uno y solo uno de ellos debe valer 1. Por ejemplo, son posibles tuplas: (0.5, 1), (1, 0.25), etc.

Conceptualmente, un objeto de tipo Truthness representa un grado de verdad sobre la ocurrencia de un evento. Si el evento no ha ocurrido, entonces ofFalse será 1 y ofTrue indicará que tan cerca estuvo de ser verdadera su ocurrencia. Del mismo modo, si el evento ha ocurrido, entonces ofTrue será 1 y ofFalse indicará que tan cerca estuvo de ser falsa su ocurrencia. Entonces, por ejemplo, un valor como (0.95, 1) indica que un evento no ocurrió, pero que estuvo muy cerca de ocurrir. En particular, el evento del que nos interesará conocer su valor de verdad es si una query devolvió resultados.

El tipo Truthness cuenta con la operación de invert(), que consiste en invertir la tupla, o sea, si la tupla es (ofTrue, ofFalse) pasará a ser (ofFalse, ofTrue).

Constantes

Las siguientes constantes serán usadas en la definición de distancia:

C : Double = 0.1
C_BETTER: Double = C + C/2

TRUE_TRUTHNESS : Truthness = Truthness(1, C)
FALSE_TRUTHNESS : Truthness = Truthness(C, 1)
FALSE_TRUTHNESS_BETTER : Truthness = Truthness(C_BETTER, 1)

Conceptualmente, TRUE_TRUTHNESS representa la verdad absoluta del tipo Truthness, donde su valor de verdad es el máximo posible y el de falsedad es el mínimo. De igual modo, FALSE_TRUTHNESS representa la falsedad absoluta, donde el valor de verdad es el mínimo posible y el de falsedad es el máximo. Notar que hay

dos constantes para representar esa falsedad en Truthness. Una de esas constantes, FALSE_TRUTHNESS_BETTER, está un poco cerca de ser verdadera que la otra, FALSE_TRUTHNESS. En general, la constante FALSE_TRUTHNESS_BETTER será usada siempre junto con FALSE_TRUTHNESS en la definición para permitir establecer dos niveles distintos de falsedad. De este modo, ante dos posibles casos excluyentes donde en ambos se debe devolver falso, se podrá premiar levemente a aquel caso que sea un poco mejor y nos acerque más a que la query devuelva resultados.

Primitivas

Las siguientes primitivas nos permitirán definir la distancia de una forma más compacta e interpretable:

Sea T = {t1, ..., tn}:

avgOfTrue(T) : Double = t1.ofTrue + ... + tn.ofTrue / |T|
anyTrue(T): Boolean = t1.ofTrue = 1 OR ... OR tn.ofTrue = 1
trueOrAvgTrue(T) : Double = IF anyTrue(T) THEN 1 ELSE avgOfTrue(T)

avgOfFalse(T) : Double = t1.ofFalse + ... + tn.ofFalse / |T|
anyFalse(T): Boolean = t1.ofFalse = 1 OR ... OR tn.ofFalse = 1
falseOrAvgFalse(T) : Double = IF anyFalse(T) THEN 1 ELSE avgOfFalse(T)

andAggregation(T): Truthness = Truthness(avgOfTrue(T), falseOrAvgFalse(T))
orAggregation(T): Truthness = Truthness(trueOrAvgTrue(T), avgOfFalse(T))

Conceptualmente, *andAggregation* representa la operación de conjunción del tipo Truthness. Por lo tanto:

andAggregation({TRUE_TRUTHNESS, TRUE_TRUTHNESS}) = TRUE_TRUTHNESS

No obstante, cuando el resultado no es TRUE_TRUTHNESS no devuelve directamente FALSE_TRUTHNESS sino que devuelve una truthness que es falsa, pero acorde a que tan cerca estuvo el resultado de ser verdadero, por ejemplo:

andAggregation({TRUE_TRUTHNESS, Truthness(0.9, 1)}) = Truthness(0.95, 1)
andAggregation({TRUE_TRUTHNESS, Truthness(0.3, 1)}) = Truthness(0.65, 1)

Análogamente, *orAggregation* representa la operación de disyunción del tipo Truthness. Por ejemplo:

orAggregation({TRUE_TRUTHNESS, FALSE_TRUTHNESS}) = TRUE_TRUTHNESS
orAggregation({FALSE_TRUTHNESS, Truthness(0.9, 1)}) = Truthness(0.5, 1)
orAggregation({FALSE_TRUTHNESS, Truthness(0.3, 1)}) = Truthness(0.2, 1)

Scaling

La siguiente función será usada frecuentemente en la definición de distancia:

scaleTrue(base, ofTrue) : Truthness = Truthness(scaleHeuristicWithBase(base, ofTrue), 1)

El código de *scaleHeuristicWithBase* se puede encontrar en el apéndice de esta tesis. Esa función tiene como objetivo proyectar un valor de ofTrue, que está entre [0, 1], en el intervalo [base, 1]. Por ejemplo, *scaleHeuristicWithBase*(0.5, 0.5) = 0.75 y *scaleHeuristicWithBase*(0.9, 0.5) = 0.95. Como se puede ver, aplicar esta función siempre resulta en el aumento de la truthness si base > 0 y no solo eso sino que además siempre será estrictamente mayor a dicha base si ofTrue > 0. Conceptualmente, se usa para premiar casos que se consideran que aumentan las chances de que la query devuelva resultados.

Definición de distancia

Dada una query, primero vamos a obtener la truthness de si dicha query devuelve resultados y la denotaremos H-Query. Una vez obtenida, la distancia estará dada por:

SQL distance = 1 - H-Query(query).ofTrue

Luego, cuanto más alto sea el valor de ofTrue (y en consecuencia, cuanto más cerca está la query de devolver resultados) menor será la distancia y viceversa. Como H-Query devuelve algo entre [0, 1], la distancia también estará en ese intervalo.

Definición de H-Query

H-Query devolverá la truthness de si una query devuelve resultados. Se define en base a H-Row-set, que devuelve la truthness de si un conjunto de filas (*row set*) posee elementos, y H-Condition, que devuelve la truthness de si una condición se cumple para un conjunto de filas.

H-Query(SELECT ... FROM <ROW-SET>) = **H-Row-set**(<ROW-SET>)

H-Query(SELECT ... FROM <ROW-SET> WHERE <CONDITION>) = andAggregation(**H-Row-set**(<ROW-SET>), **H-Condition**(<CONDITION>, <ROW-SET>))

H-Query(<QUERY1> UNION [ALL] <QUERY2>) = orAggregation(**H-Query**(<QUERY1>), **H-Query**(<QUERY2>))

Definición de H-Row-set

H-Row-set devolverá la truthness de si un conjunto de filas posee elementos. Se usa para premiar casos donde la query no da resultados, pero las tablas están completa o parcialmente populadas, lo cual aumenta las chances de que alguna entrada satisfaga la cláusula **WHERE**. Se define como:

H-Row-set(<ROW-SET>) = IF <ROW-SET> is a single table THEN
H-Table(<ROW-SET>)
ELSE IF <ROW-SET> is a join
H-Join(<ROW-SET>)
ELSE IF <ROW-SET> is a subquery
H-Query(<ROW-SET>)
ELSE //FROM clause was not present
TRUE_TRUTHNESS

De esta manera, si las filas vienen dadas por una tabla usará H-Table, que devuelve la truthness de si una tabla tiene filas, si las filas vienen dadas por un **JOIN**, usará H-Join, que devuelve la truthness de si un JOIN devuelve filas y si las filas vienen dadas por una subquery, usará H-Query, que ya vimos que devuelve la truthness de si una query devuelve filas. Por último, para las queries sin **FROM** (por ejemplo **SELECT 1)** se devuelve TRUE_TRUTHNESS ya que estas queries siempre devuelven resultados.

Definición de H-Table

H-Table devolverá la truthness de si una tabla tiene filas. Se define como:

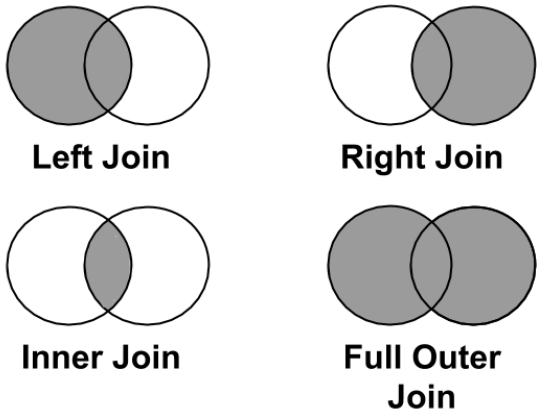
H-Table(<TABLE>) = getTruthnessToEmpty(n).invert(), donde n es la cantidad de filas que tiene <TABLE>

El código de *getTruthnessToEmpty* se puede encontrar en el apéndice de esta tesis. Esta función devuelve una truthness verdadera cuando la tabla está vacía y una falsa cuando no. Como la estamos invirtiendo, H-Table retornará una truthness verdadera sólo cuando la tabla no esté vacía.

Definición de H-Join

H-Join devolverá la truthness de si un **JOIN** devuelve resultados. La definición depende del tipo de **JOIN**.

Para entender mejor las definiciones que se darán a continuación es útil primero representar los **JOIN** como diagramas de Venn. En estos diagramas, los conjuntos serán de filas (o sea, lo que venimos llamando row-set como una tabla o una subquery) y en la intersección se hallarán las filas que cumplen la condición del join:



- **Inner join:**

  En el **INNER JOIN** se seleccionan sólo aquellas filas que fueron joineadas. Luego, para que devuelva filas hace falta que ambos conjuntos de filas no estén vacíos y luego que alguna fila en el producto cartesiano entre ambos conjuntos cumpla la condición del join. Entonces:

  H-Join(<ROW-SET1> INNER JOIN <ROW-SET2> ON <CONDITION>) = andAggregation(**H-Row-set**(<ROW-SET1>), **H-Row-set**(<ROW-SET2>), **H-Condition**(<CONDITION>, <ROW-SET1> x <ROW-SET2>))

  Notar que el producto cartesiano se implementa mediante un CROSS JOIN.

- **Left outer join:**

  En el **LEFT [OUTER] JOIN** se seleccionan las filas que fueron joineadas y también aquellas filas del conjunto de la izquierda que no pudieron ser joineadas. Luego, para que devuelva filas alcanza con que el conjunto de la izquierda no esté vacío. Entonces:

  H-Join(<ROW-SET1> LEFT [OUTER] JOIN <ROW-SET2> ON <CONDITION>) = **H-Row-set**(<ROW-SET1>)

Notar que el keyword **OUTER** es opcional y que los campos no joineados del conjunto de filas de la derecha se rellenan con **NULL.** Este comportamiento será igual para todos los **JOIN** de tipo **OUTER**.

- **Right outer join:**

En el **RIGHT [OUTER] JOIN** se seleccionan las filas que fueron joineadas y también aquellas filas del conjunto de la derecha que no pudieron ser joineadas. Luego, para que devuelva filas alcanza con que el conjunto de la derecha no esté vacío. Entonces:

H-Join(<ROW-SET1> RIGHT [OUTER] JOIN <ROW-SET2> ON <CONDITION>) = **H-Row-set**(<ROW-SET2>)

- **Full outer join:**

En el **FULL [OUTER] JOIN** se seleccionan las filas que fueron joineadas y también aquellas filas que no pudieron ser joineadas, tanto del conjunto de filas de la izquierda como el de la derecha. Luego, para que devuelva filas alcanza con que al menos uno de los dos conjuntos no esté vacío. Entonces:

H-Join(<ROW-SET1> FULL [OUTER] JOIN <ROW-SET2> ON <CONDITION>) = orAggregation(**H-Row-set**(<ROW-SET1>), **H-Row-set**(<ROW-SET2>))

- **Cross join:**

En el **CROSS JOIN** se seleccionan las filas resultantes del producto cartesiano de los conjuntos de filas, o sea, todas las filas de un conjunto contra todas las filas del otro conjunto. Luego, recordando que el producto cartesiano entre un conjunto C y ∅ es ∅, para que devuelva filas hace falta que ninguno de los dos conjuntos esté vacío. Entonces:

H-Join(<ROW-SET1> CROSS JOIN <ROW-SET2>) = andAggregation(**H-Row-set**(<ROW-SET1>), **H-Row-set**(<ROW-SET2>))

Definición de H-Condition

Devuelve la truthness de si una condición se cumple para un conjunto de filas.

Sea maxOfTrue = max { δ_r(<CONDITION>).ofTrue | r ∈ <ROW-SET> }

H-condition(<CONDITION>, <ROW-SET>) =
    IF <ROW-SET> is empty THEN
        FALSE_TRUTHNESS
    ELSE
        IF maxOfTrue == 1
        THEN TRUE_TRUTHNESS
        ELSE scaleTrue(C, maxOfTrue)

Queda claro que es mejor el caso de tener un conjunto de filas que no cumplen la condición que no tener ni siquiera un conjunto de filas sobre los que aplicar la condición. Al utilizar la función scaleTrue sobre maxOfTrue (última línea) nos estamos asegurando que, sea cual sea el valor de maxOfTrue, la truthness resultante de tener filas que no cumplen será siempre mayor a la truthness de cuando no hay filas en absoluto (FALSE_TRUTHNESS). Esto ocurre porque el ofTrue de FALSE_TRUTHNESS es C y el ofTrue de scaleTrue(C, maxOfTrue) será mayor estricto a C, ya que como vimos será la proyección de maxOfTrue entre C y 1.

Definición de δr

Esta función toma una condición SQL y devuelve la truthness de si dicha condición se cumple para la fila r. Primero, hace una asignación de las variables libres de la condición según los valores de la fila r y luego llama a la función δ genérica. Por ejemplo, supongamos que la condición es “salary >= 2000” y que las filas son las siguientes:

Employees	
ID	Salary
1	1500
2	1900

Luego:

δ_1(salary >= 2000) = $\delta_{salary = 1500}$(salary >= 2000) = δ(1500 >= 2000)
δ_2(salary >= 2000) = $\delta_{salary = 1900}$(salary >= 2000) = δ(1900 >= 2000)

Definición de δ

Esta función toma una condición SQL y devuelve la truthness de si dicha condición se cumple. Es análoga a la branch distance, pero para condiciones escritas en lenguaje SQL.

Operadores de comparación

- δ(x Θ y) = IF x == NULL AND y == NULL
  THEN FALSE_TRUTHNESS
  ELSE IF x == NULL OR y == NULL
  THEN FALSE_TRUTHNESS_BETTER
  ELSE scaleTrue(C_BETTER, (x $\Theta_{type(x,y)}$ y).ofTrue)

- x $=_{num}$ y = getEqualityTruthness(x, y)

- x $\neq_{num}$ y = getEqualityTruthness(x, y).invert() //a != b => !(a == b)

- x $>_{num}$ y = getLessThanTruthness(y, x) //a > b => b < a

- x $\geq_{num}$ y = getLessThanTruthness(x, y).invert() //a >= b => ! (a < b)

- x $<_{num}$ y = getLessThanTruthness(x, y)

- $x \leq_{num} y$ = getLessThanTruthness(y, x).invert() //a <= b => b >= a => !(b < a)

- $x =_{string} y$ = getStringEquals(x, y)

- $x \neq_{string} y$ = getStringEquals(x, y).invert()

- $x \Theta_{bool} y$ = toInt(x) Θ_{num} toInt(y) donde toInt convierte un booleano en un entero (usando b != 0 ? true : false) y Θ puede ser cualquier operador en el conjunto { =, ≠ }

- $x \Theta_{date} y$ = toLong(x) Θ_{num} toLong(y) donde toLong convierte un date en un long (usando la función getTime() de Java) y Θ puede ser cualquier operador en el conjunto { =, ≠, >, ≥, <, ≤ }

El código de las funciones *getEqualityTruthness*, *getLessThanTruthness* y *getStringEquals* se puede encontrar en el apéndice de esta tesis. Similarmente a lo que ocurre con la branch distance, la truthness que devuelven estas funciones aumenta a medida que las variables se acercan a hacer verdadera la condición.

Operadores booleanos

- δ(A AND B) = andAggregation(δ(A), δ(B))

- δ(A OR B) = orAggregation(δ(A), δ(B))

- δ(NOT A) = δ(A).invert()

Operadores de chequeo de booleanos

- δ(x IS TRUE) = IF x == NULL OR !x
  THEN FALSE_TRUTHNESS
  ELSE TRUE_TRUTHNESS

- δ(x IS NOT TRUE) = δ(x IS FALSE)

- δ(x IS FALSE) = IF x == NULL OR x
  THEN FALSE_TRUTHNESS
  ELSE TRUE_TRUTHNESS

- δ(x IS NOT FALSE) = δ(x IS TRUE)

Operadores de chequeo de null

- δ(x IS NULL) = IF x == NULL
  THEN TRUE_TRUTHNESS
  ELSE FALSE_TRUTHNESS

- δ(x IS NOT NULL) = δ(x is NULL).invert()

Operadores de rango

- δ(x BETWEEN A and B) = δ(x >= A AND x <= B)

- δ(x NOT BETWEEN A and B) = δ(x < A OR x > B)

Operadores de subquery

- δ(EXISTS <QUERY>) = H-query(<QUERY>)

- ANY:

  Sea maxOfTrue = max { δ(x Θ y).ofTrue | y ∈ <QUERY> }

  δ(x Θ ANY <QUERY>) =
  IF <QUERY> is empty
  THEN FALSE_TRUTHNESS
  ELSE
  IF maxOfTrue == 1
  THEN TRUE_TRUTHNESS
  ELSE scaleTrue(C, maxOfTrue)

- ALL:

  Sea minOfTrue = min { δ(x Θ y).ofTrue | y ∈ <QUERY> }

  δ(x Θ ALL <QUERY>) =
  IF <QUERY> is empty
  THEN TRUE_TRUTHNESS
  ELSE
  IF minOfTrue == 1
  THEN TRUE_TRUTHNESS
  ELSE scaleTrue(C, minOfTrue)

- δ(x IN <QUERY>) = δ(x = ANY <QUERY>)

- δ(x NOT IN <QUERY>) = δ(x ≠ ALL <QUERY>)

- δ(x IN <VALUES>) = δ(x = ANY <VALUES>)

- δ(x NOT IN <VALUES>) = δ(x ≠ ALL <VALUES>)

Features

En esta sección se hará un relevamiento de los features de la distancia SQL avanzada. Aparte de soportar subqueries y poder dar gradiente a partir de la cláusula FROM, la distancia avanzada también soporta nuevos operadores, tipos de datos, etc. Además, soporta conversiones entre tipos de datos bastante comunes en las tecnologías SQL.

Tipo numérico

Se soportan variables de los siguientes tipos: Double, Long y Signed expression.

Tipo string

Se soportan variables de tipo String.

Tipo fecha

Se soportan variables de los siguientes tipos: Date, Time y Timestamp. La distancia avanzada SQL admite los literales **DATE**, **TIME** y **TIMESTAMP**:

```
created >= DATE '1968-01-01'
created >= TIMESTAMP '1968-01-01 00:00:00.1'
created >= TIME '00:00:01'
```

Y también admite los literales {d}, {t} y {ts}:

```
created >= {d '1968-01-01'}
created >= {ts '1968-01-01 00:00:00.1'}
created >= {t '00:00:01'}
```

Dado que muchas tecnologías de SQL lo soportan, en la distancia SQL avanzada se implementaron las comparaciones entre fechas y strings. Ahora es posible comparar un date y una string usando cualquier operador de comparación (equals, not equals, greater than, etc.) y también operadores como **BETWEEN**, **IN**, etc:

```
created >= '1968-01-01'
created BETWEEN '1968-01-01' AND '1969-01-01'
```

Tipo booleano

Supongamos que tenemos una columna booleana `approved`. Se soportan expresiones de este tipo:

```
approved = true AND ...
```

Sin embargo, en muchas bases de datos como MySQL no existe el tipo boolean sino que en cambio usan enteros como **tinyint**. Por eso, fue necesario implementar una conversión entre enteros y booleanos para que funcione el caso anterior ya que en la columna `approved` vendrá un **1** en vez de **true**.

El criterio de conversión entre números y booleanos es el mismo que se emplea en las bases de datos y en la programación en general, un número es **false** si y sólo si es cero.

Obviamente también funcionarán expresiones de este tipo ya que no es necesaria ninguna conversión:

```
approved = 1 AND ...
```

Además, en la distancia SQL extendida se agregó la posibilidad de usar la columna directamente como expresión booleana:

```
approved AND ...
```

Operadores de comparación

Se soportan los operadores de comparación equals, not equals, greater than, greater than or equals, minor than y minor than or equals para los números y las fechas. Para las string y los booleanos, sólo se soportan el equals y el not equals.

Operadores aritméticos y paréntesis

La distancia SQL extendida soporta expresiones aritméticas y paréntesis. Esto hace posible poder procesar expresiones como:

```
number - 1 >= (other_number / 2 + 3) * 4
```

Operadores booleanos

Se soportan los operadores **AND** y **OR**.

Operadores de chequeo booleanos

Se soportan los operadores **IS TRUE** y **IS FALSE** y sus contrapartes **IS NOT TRUE** y **IS NOT FALSE**, respectivamente.

Operadores de chequeo de null

Se soporta el operador **IS NULL** y su contraparte **IS NOT NULL**.

Operador de rango

Se soporta el operador **BETWEEN** tanto para números como para fechas. Es decir, se admiten expresiones como:

```
age BETWEEN 22 AND 28
```
y
```
created BETWEEN DATE '1968-01-01' AND DATE '1970-01-01'
```

Recordemos que como se implementó la comparación entre fechas y strings también se admiten expresiones como:

```
created BETWEEN '1968-01-01' AND '1970-01-01'
```

O incluso:

```
created BETWEEN DATE '1968-01-01' AND '1970-01-01'
```

También se soporta el operador **NOT BETWEEN** para los casos descriptos anteriormente.

Subqueries

Se soportan todos los casos expuestos en la sección de “Limitaciones anteriores”, es decir: subqueries en el FROM, operadores de subqueries, subqueries como valor y unión de subqueries.

Funciones

Se soportan expresiones que posean built-in functions como **UPPER**, **LOWER**, etc. Para poder dar soporte a cualquier built-in function de cualquier tecnología SQL, se pide de antemano el resultado de la función en el **SELECT**. Por ejemplo, si tenemos la query:

```
SELECT salary FROM employees WHERE UPPER(name) = 'john' AND salary = 8000
```

Entonces la query que se usará para calcular la distancia a las filas será:

```
SELECT salary, UPPER(name) FROM employees
```

Por último, se procede a calcular la distancia como siempre tomando a UPPER(name) como una columna más.

Otros statements

Así como nos interesa que los **SELECT** devuelvan filas, también nos interesará que los **DELETE** y los **UPDATE** apliquen sobre filas existentes. Para poder calcular la distancia en estos casos, ambos statements se pueden convertir a un **SELECT**. Por ejemplo, si tenemos el **DELETE**:

```
DELETE FROM employees WHERE name = 'john'
```

Este puede ser convertido a:

```
SELECT * FROM employees WHERE name = 'john'
```

De ahí en más, se calcula la distancia normalmente.

Para los **UPDATE** el procedimiento es el mismo.

Testing

Dado que la tesis trata esencialmente de testing y con el objetivo de desmitificar el dicho "en casa de herrero, cuchillo de palo", para la implementación de la SQL distance se escribieron 155 tests (89 tests de unidad y 66 tests de integración) que alcanzan un 91% de coverage.

Para mejorar la mantenibilidad de los tests ante cambios futuros en la definición, las aserciones en los tests se llevaron a cabo usando una DSL (domain specific language) que permite describir el resultado de la query sin tener que indicar el número, por ejemplo:

```
execute("INSERT INTO customers VALUES (1, 'john', 25)");
Truthness truthness = calculate("SELECT * FROM customers WHERE
age = 24.0");
assertEquals(singleRowWith(singleCondition(eq(25, 24))),
truthness);
```

Otros enfoques

En el trabajo "Search-based test data generation for SQL queries", J. Castelein et al. no buscaban probar código sino queries, sin embargo, como parte de su solución también requirieron de una distancia entre una query SQL y las filas de una tabla.

A grandes rasgos, su enfoque consistió en ejecutar la query en una base de datos interna y luego analizar el árbol que describe el plan físico de la query. Si la query devuelve resultados, el plan se ejecuta hasta la raíz del árbol y la distancia es cero. Pero si no devuelve resultados, el plan se detendrá en algún nivel del árbol y ese nivel del árbol será tomado como la distancia.

La ventaja de este enfoque es que no hay que implementar casi nada (usaron el código de HSQLDB con algunas líneas modificadas), pero cuenta con algunas desventajas. Una desventaja es la eficiencia porque requiere copiar cada vez los

schemas y las tablas completas (que pueden ser grandes) a la base de datos interna. La eficiencia es especialmente importante en este contexto porque el tiempo para realizar la búsqueda es limitado y cualquier overhead implica explorar menos el espacio de soluciones. Otra desventaja es la compatibilidad requerida entre la base de datos interna y la base de datos de la aplicación, es habitual ver diferencias entre tipos de datos, built-in functions, etc. de las diferentes tecnologías SQL (MySQL, PostgreSQl, MariaDB, SQL Server, etc.).

Experimentos

Para que una API pueda ser probada por EvoMaster, se requiere wrappearla en lo que se denomina un *controller*. Este controller permite manipular la API a los fines de poder correr en ella los tests que se van generando. Para eso, el controller cuenta con métodos para levantar la API al principio, para resetear el estado de la API después de un test, para obtener las métricas de coverage (y las queries SQL con resultados vacíos), para bajar la API al final, etc.

Los experimentos fueron realizados usando EvoMaster Benchmark (EMB), que es un conjunto de APIs tomadas del mundo real y convenientemente wrappeadas para ser probadas por EvoMaster. Para probar las distancias nos restringiremos sólo a aquellas APIs de EMB que usan una base de datos y que además esta es de tipo SQL.

Se configuró EvoMaster para usar 1 hora por corrida (--maxTime 1h) y para permitirle hacer inserciones SQL (--generateSqlDataWithSearch true). Con la configuración anterior fija, se hicieron 5 corridas usando la distancia avanzada (--heuristicsForSQLAdvanced true) y 5 corridas usando la distancia estándar (--heuristicsForSQLAdvanced false).

El análisis de las corridas para cada distancia se realizó conforme a lo propuesto en [10], donde para algoritmos como MIO se aconseja aplicar el Test U de Mann-Whitney y la medida Â12 de Vargha-Delaney. Por un lado, el Test U de Mann-Whitney determina si existen diferencias significativas entre ambas distribuciones. De este test obtenemos un p-valor, la probabilidad de observar los resultados obtenidos si la hipótesis nula (ambas distribuciones son iguales) fuera cierta. Como referencia, un p-valor menor a 0,05 indica que se rechaza la hipótesis nula, es decir, que las distribuciones son distintas. Por otro lado, la medida Â12 de

Vargha-Delaney cuantifica la magnitud de esta diferencia. Como referencia, Â12 >= 0,56 es una diferencia pequeña, Â12 >= 0,64 es mediana y Â12 >= 0, 71 es grande.

En la siguiente tabla se encuentran los resultados obtenidos para cada API en EMB, donde se indica la media de la distancia avanzada, la media de la distancia estándar y los p-valores y la medida Â12 de la comparación de las distribuciones:

API	Avanzada	Estándar	p-value	Â12
scout	923	930	1.0	0.53
features-service	347.2	354	0.08	0.9
cwa-verification	417.4	427.6	0.67	0.4
proxyprint	1448.3	1430.5	0.8	0.67
catwatch	845.4	791	0.15	0.2
news	122.2	122	1.0	0.48
market	540.4	549	0.47	0.7

Pese a que la distancia avanzada es capaz de procesar más tipos de queries que la distancia estándar, no se observó en los experimentos que haya una diferencia estadísticamente significativa en los resultados ya que en ninguna se puede rechazar la hipótesis nula de que ambas distribuciones son iguales.

Queries resueltas

Surge la pregunta de cuántas queries adicionales efectivamente resuelve la distancia avanzada para las APIs presentes en EMB. En la siguiente tabla se encuentra para cada API en EMB cual fue el porcentaje de queries que solo la distancia avanzada resolvió, el porcentaje de queries que ambas distancias resolvieron y el porcentaje de queries que ninguna distancia fue capaz de resolver (no se reportaron casos que sólo la distancia estándar pudo resolver):

API	Sólo avanzada	Ambas distancias	Ninguna distancia
scout	0.76%	95.41%	3.84%
features-service	30.91%	64.30%	4.79%
cwa-verification	0%	100%	0%
proxyprint	0%	100%	0%
catwatch	0%	100%	0%
news	0.02%	99.98%	0%
market	0%	100%	0%

Se puede apreciar que el porcentaje de queries que solo la distancia avanzada pudo resolver en general no es muy grande para las APIs de EMB. Además, se analizó para cada API en particular cuáles eran esas queries que solo la distancia avanzada pudo resolver. Se observó que en general eran siempre la misma query, pero instanciada diferente. Por ejemplo, para la API features-service:

```
SELECT … FROM (…) features WHERE features.for_product_id = 1
SELECT … FROM (…) features WHERE features.for_product_id = 3
…
```

Todas estas queries provienen del mismo lugar del código y a lo sumo pueden destrabar los mismos branches al devolver algún resultado. Entonces, pese a su porcentaje sobre el total, no hacen una diferencia notable en el resultado final.

Vale la pena destacar que se analizaron en detalle aquellas queries que ninguna de las dos distancias pudo resolver en busca de oportunidades de mejoras, pero mayormente eran queries irresolubles, por ejemplo porque la tabla no existe.

Nuevos features

El bajo porcentaje de queries que solo la distancia avanzada pudo resolver se debe a que los nuevos features soportados están presentes en muy pocas queries. Por ejemplo, solo la API features-service cuenta con queries que contienen subqueries (alcanzando un 35,41% del total, pero ya vimos que en realidad eran siempre las mismas queries repetidas). El hecho de que pocas queries contengan subqueries es esperable en cierta forma ya que que en la actualidad la mayoría de las queries presentes en las APIs ya no son escritas por los desarrolladores sino que son generadas automáticamente por los frameworks (Hibernate, Spring Data, etc.) y, por cuestiones de eficiencia, en la generación automática se prefiere el uso de **JOIN** por sobre el uso de subqueries. Otros nuevos features implementados como operadores y tipos de datos tampoco estaban presentes o tenían una frecuencia despreciable.

Gradiente adicional

Hemos visto que, en los casos donde una query no da resultados aún sin la cláusula **WHERE,** la distancia avanzada es capaz de dar gradiente a partir de la cláusula **FROM**. Es decir, puede determinar que tan cerca está el **FROM** de devolver resultados, lo cual es una condición que nos acerca a que la query entera devuelva resultados.

A diferencia de los nuevos features implementados, donde necesitamos que las queries contengan esos features para poder sacar algún provecho, este gradiente adicional puede ser aplicado a cualquier query. No obstante, tampoco hizo una diferencia porque cuando las inserciones SQL están activadas ocurre que las queries sin el **WHERE** frecuentemente ya devuelven resultados, es decir, el **FROM** ya trae resultados. En esos casos no se usa en absoluto el gradiente sobre el **FROM** porque esta cláusula ya fue satisfecha y la distancia pasa a depender enteramente del **WHERE**. Si bien podría ser útil dar gradiente sobre el **FROM** cuando la inserción SQL está desactivada, lo cierto es que no hay muchos motivos para desactivarla ya que en general se obtienen mejores resultados cuando está activada.

Conclusión

Esta nueva distancia contribuyó a la completitud de la solución ya que es capaz de resolver más tipos de queries, sin embargo, estas queries no tienen una frecuencia suficientemente grande en EMB para lograr una diferencia significativa en los resultados. Luego, estos resultados quedaron más condicionados por factores aleatorios (las soluciones iniciales, las mutaciones, etc. del algoritmo evolutivo) que por cualquier contribución que podría haber hecho la nueva distancia. Hará falta incorporar más APIs a EMB para poder ver el potencial de la nueva distancia, lo cual no es un proceso sencillo porque como se vio cada API debe ser wrappeada manualmente para poder ser probada por EvoMaster. Por último, se descubrió también que cuando las inserciones SQL están activadas el gradiente del **FROM** no suele proveer demasiada información adicional a la distancia.

Apéndice

Implementación de funciones usadas en la definición de la SQL distance:

```
public static double scaleHeuristicWithBase(double base, double
heuristic){

  if(heuristic < 0 || heuristic >= 1){
    throw new IllegalArgumentException("Invalid heuristic: " +
base);
  }

  if(base < 0 || base >= 1){
    throw new IllegalArgumentException("Invalid base: " + base);
  }

  return base + ((1 - base) * heuristic);
}

public static Truthness getTruthnessToEmpty(int len) {
  if (len < 0){
    throw new IllegalArgumentException("Invalid length " + len);
  }

  Truthness t;
  if (len == 0) {
    t = TRUE_TRUTHNESS;
  } else {
    t = new Truthness(1d / (1d + len), 1);
  }

  return t;
}

public static Truthness getEqualityTruthness(int a, int b) {

  double distance = getDistanceToEquality(a, b);
  double normalizedDistance = normalizeValue(distance);

  return new Truthness(
```

```
    1d - normalizedDistance,
    a != b ? 1d : 0d
  );
}

public static double getDistanceToEquality(double a, double b) {

  if (!Double.isFinite(a) || !Double.isFinite(b)) {
    return Double.MAX_VALUE;
  }

  double distance;
  if (a < b) {
    distance = b - a;
  } else {
    distance = a - b;
  }

  if (distance < 0 || !Double.isFinite(distance)) {
    //overflow has occurred
    return Double.MAX_VALUE;
  } else {
    return distance;
  }
}

public static double normalizeValue(double v) {
  if (v < 0) {
    throw new IllegalArgumentException("Negative value: " + v);
  }

  if(Double.isInfinite(v) || v == Double.MAX_VALUE){
    return 1d;
  }

  return v / (v + 1d);
}

public static Truthness getLessThanTruthness(long a, long b) {
  double distance = getDistanceToEquality(a, b);
  return new Truthness(
```

```
    a < b ? 1d : 1d / (1.1d + distance),
    a >= b ? 1d : 1d / (1.1d + distance)
  );
}

public static Truthness getStringEqualityTruthness(String str1,
String str2){
  if (str1.equals(str2)) {
    return TRUE_TRUTHNESS;
  } else {
    final double base = C;
    double distance = getLeftAlignmentDistance(str1, str2);
    double h = heuristicFromScaledDistanceWithBase(base,
distance);
    return new Truthness(h, 1d);
  }
}

public static long getLeftAlignmentDistance(String a, String b)
{

  long diff = Math.abs(a.length() - b.length());
  long dist = diff * 65_536;

  for (int i = 0; i < Math.min(a.length(), b.length()); i++) {
    dist += Math.abs(a.charAt(i) - b.charAt(i));
  }

  if(dist < 0){
    dist = Long.MAX_VALUE; // overflow
  }

  return dist;
}

public static double heuristicFromScaledDistanceWithBase(double
base, double distance){

  if(base < 0 || base >= 1){
    throw new IllegalArgumentException("Invalid base: " + base);
  }
```

```
  if(distance < 0){
    throw new IllegalArgumentException("Negative distance: " +
distance);
  }

  if(Double.isInfinite(distance) || distance ==
Double.MAX_VALUE){
    return base;
  }

  return base + ((1 - base) / (distance + 1));
}
```

Bibliografía

[1] A. Arcuri, J.P. Galeotti. Handling SQL Databases in Automated System Test Generation. ACM Transactions on Software Engineering and Methodology (TOSEM). 2020.

[2] A. Arcuri, J.P. Galeotti. SQL Data Generation to Enhance Search-Based System Testing. ACM Genetic and Evolutionary Computation Conference (GECCO). 2019.

[3] A. Arcuri. RESTful API Automated Test Case Generation with EvoMaster. ACM Transactions on Software Engineering and Methodology (TOSEM). 2019.

[4] J. Castelein, M. Aniche, M. Soltani, A. Panichella, A. van Deursen. Search-based test data generation for SQL queries. ACM/IEEE International Conference on Software Engineering (ICSE). 2018.

[5] A. Arcuri. Test Suite Generation with the Many Independent Objective (MIO) Algorithm. Information and Software Technology (IST). 2018.

[6] A. Arcuri. EvoMaster: Evolutionary Multi-context Automated System Test Generation. IEEE International Conference on Software Testing, Validation and Verification (ICST). 2018.

[7] A. Arcuri. RESTful API Automated Test Case Generation. IEEE International Conference on Software Quality, Reliability & Security (QRS). 2017.

[8] A. Arcuri. Many Independent Objective (MIO) Algorithm for Test Suite Generation. Symposium on Search-based Software Engineering (SSBSE). 2017.

[9] A. Panichella, F. Kifetew, P. Tonella. Automated test case generation as a many objective optimization problem with dynamic selection of the targets. IEEE Transactions on Software Engineering (TSE). 2017.

[10] A. Arcuri, L. Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. Software Testing, Verification and Reliability (STVR). 2014.

[11] G. Fraser, A. Arcuri. Whole Test Suite Generation. IEEE Transactions on Software Engineering. 2013.

[12] M. Harman, S. A. Mansouri, Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR). 2012.

[13] B. Korel. Automated Software Test Data Generation. IEEE Transactions on Software Engineering. 1990.