Documentación práctica 2.

Se necesita un framework de Excepciones para manejar las distintas situaciones anómalas (Excepciones) de la librería para el menejo de sentencias SQL (Práctica 1). Se deberán establecer excepciones marcadas y no marcadas dentro del framework.

Del mismo modo se deberá implementar un sistema interactivo (interprete interactivo) para consultar tablas (archivos CSV) de diferentes bases de datos (carpetas). Del mismo modo se deberán agregar funciones *BUIT-IN* (Ver anexo).

¿Cómo funciona?

El programa funciona como un intérprete de sentencias **SQL**, que actualmente soporta las sentencias:

- `SHOW TABLES;`
- `DROP TABLE [NOMBRE TABLA];`
- `CREATE TABLE [NOMBRE TABLA]([NOMBRE] [TIPO] [NULL/NOT NULL] [PRIMARY KEY/""]...);`
- `INSERT INTO [NOMBRE TABLA] (NAME1, NAME2, NAME3,...) VALUES (VAL1, VAL2, VAL3,...);`
- `UPDATE [NOMBRE TABLA] SET [COLUMNA [OPERADOR] [VALOR], COLUMNA [OPERADOR] [VALOR],...] WHERE [CONDICIÓN];`
- `SELECT [COLUMNAS [AS 'ALIAS']/*] FROM [NOMBRE TABLA] WHERE [CONDICIÓN];`
- `DELETE FROM [NOMBRE TABLA] WHERE [CONDICIÓN];`
- `USE [DATABASE PATH];`

Todas estas funciones están diseñadas para ser *case insensitive*, **excepto** los nombres de tablas, que si requieren ser escritos tal cuál fueron ingresados en primer lugar.

Al ingresar una query, está misma ingresa al método `parseQuery` de la clase `Parser.java`, una vez allí, un *if-else* redirige la query a su respectiva función que la va a manejar, buscando coincidencias con algunas palabras que se mantienen estáticas en todas las consultas. Este primer paso es solamente para un primer chequeo de la consulta, sin embargo, la revisión completa de las mismas se realiza dentro de la función especializada para ello, que vamos a revisar en breve.

USE

El comando `use` es, considero yo, el más sencillo de todos, se encarga solamente de verificar que la ruta ingresada por el usuario exista y que, además, se pueda escribir en ella. Está tarea la realiza *escribiendo* un archivo **temporal** en la ruta ingresada para comprobar que se pueda escribir en ella.

```
File tempFile = File.createTempFile("writeTest", ".tmp", new File(path));
tempFile.delete();
```

SHOW TABLES

El comando `show tables` es también bastante sencillo, se basa solamente en verificar que estemos actualmente en una base de datos y, una vez allí, listar todas las bases de datos (archivos .csv) que tengamos en dicho directorio.

```
for (File file : files) {
    String name = file.getName();
    if (name.contains("aux"))
        continue;
    if (file.getName().contains(".csv") && !file.getName().contains("~"))
        System.out.println("* " + name.substring(0, name.indexOf(".csv")));
}
```

DROP TABLE

El comando `drop table` se centra en eliminar una base de datos del sistema, basta con escribir `drop table [NOMBRE TABLA]` para borrarla del directorio (base de datos) en el que nos encontramos. Además, al ser un comando permanente, se le presenta al usuario la opción de decidir si borrar o no el directorio.

```
for (File file : files)
    if (file.getName().equalsIgnoreCase(name)) {
        System.out.println("¿Seguro que deseas borrar la tabla(y/n)?");

        String answ;
        try {
            answ = bf.readLine();

        if (answ.equalsIgnoreCase("y")) {
            file.delete();
            new File(FileManagement.getDatabasePath() + auxFile).delete();
            return "Tabla borrada éxitosamente";
        } else {
            return "Tabla no eliminada";
        }

        } catch (IOException e) {
            throw new IOException("No se pudo borrar la tabla");
        }
    }
```

SELECT

El comando `select` es posiblemente el más complejo de todos, no tanto por la clase en si, sino que la correcta implementación del mismo era vital para el funcionamiento de mi sistema, pues novedades como la sentencia condicional **where** eran totalmente necesarias en mi programa.

Al tener una funcionalidad tan compleja, decidí realizar una clase especial, solamente para está sentencia, que es `select.java`.

Esta sentencia tiene en cuenta **alias** de columnas, así como también es capaz de realizar operaciones aritméticas y anidadas.

WHERE

Para desarrollar la condicional `where`, mi pensamiento inicial fue hacerlo iterativo, de manera que solamente uniera las condicionales con sus operadores lógicos, sin embargo, más temprano que tarde me di cuenta que, para mi mala fortuna, es posible también anidar las sentencias **where**, lo que hacia que mi solución iterativa sirviése solamente cuando no se ingresaban paréntesis en la sentencia. Para solucionar esto, trabaje con una solución **recursiva**, que se basa en solucionar primero lo que se encuentra entre paréntesis, antes de pasar a lo que no.

Ejemplo:

Si tuviesemos una condicional, por ejemplo, `id = 1 or (name = 'manuel' and (money <= 1500 or id > 1))`, esta se resolvería de la siguiente manera:

1. `id = 1 or (name = 'manuel' and (money <= 1500 or id > 1))`
2. `id = 1 or (name = 'manuel' and false)`
3. `id = 1 or false`
4. `true or false`
5. `true`

Eval

Dentro de mi programa, en específico dentro de la claúsula **select**, se hace uso de una clase llamada `Eval.java`. Está clase es esencial, pues evalúa las sentencias ingresadas por el usuario dentro de la columna select. Las funciones que soporta son las siguientes:

- `ROUND(NUMERO/EXPRESION)`
- `CEIL(NUMERO/EXPRESION)`
- `FLOOR(NUMERO/EXPRESION)`
- `UCASE(CADENA)`
- `LCASE(CADENA)`
- `CAPITALIZE(CADENA)`
- `COUNT(COLUMNA NUMERICA/*)`
- `AVG(COLUMNA NUMERICA)`
- `SUM(COLUMNA NUMERICA)`
- `MAX(COLUMNA)`
- `MIN(COLUMNA)`
- `RAND(VALOR LIMITE/'')`

Donde las funciones que realizan alguna operación aritmética(round, ceil y floor) son anidables, de manera que puedes realizar sentencias como la siguiente:

```
select ceil(id + 2 + round(money / 500)) as 'operacion compleja' from test;
```

La manera de resolver estas sentencias es prácticamente la misma que la sentencia **where**, hago la evaluación, de los paréntesis más anidados, a aquellos exteriores, de manera que la sentencia anterior, tomando `id = 1` y `money = 500`, se operaría de la siguiente manera:

1. `ceil(1 + 2 + round(500/500))`
2. `ceil(1 + 2 + 1)`
3. `ceil(4)`
4. `4`

Pero, ¿quién realiza estas operaciones aritméticas?

EvaluateExpression

Está clase también es bien importante, pues se encarga de realizar las operaciones aritméticas necesarias para el funcionamiento del programa. En realidad, la clase es una implementación (inyectada y con esteroides) del algoritmo `Shunting Yard`, que además de realizar las operaciones básicas(+-*/) realiza también operaciones como división entera(div) y módulo(%).

Este algoritmo toma en cuenta la prioridad de los operadores, respetando la jerarquía de operaciones y los paréntesis:

```
private static int precedence(char op) {
        switch (op) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
            case '%':
            case '#':
                return 2;
            default:
                return -1;
        }
    }
```

Otras claúsulas

Las claúsulas que quedan son realmente son solamente una modificación de las clases sobre las que crece mi programa(`EvaluateExpression.jav`, `Eval.java` y `Where.java`), pues sentencias como la sentencia `UPDATE` admite operaciones aritméticas en su claúsula `SET`, así como también admite condicionales(`WHERE`).

Test cases

¿Es palabra reservada?

Este primer **test case** muestra la efectividad de la función `isReservedWord` de la clase `Utilities.java`, necesaria debido a que nombres de columnas o nombres de tablas no pueden ser llamadas como una

palabra reservada pues podría provocar *errores* y *excepciones* inesperadas.

```
@Test
    public void testIsReservedWord_True() {
        FileManagement.initialValidations();

        assertTrue(Utilities.isReservedWord("SELECT"));
        assertTrue(Utilities.isReservedWord("INSERT"));
        assertTrue(Utilities.isReservedWord("UPDATE"));
        assertTrue(Utilities.isReservedWord("FROM"));
        assertTrue(Utilities.isReservedWord("INT"));
        assertTrue(Utilities.isReservedWord("VARCHAR"));
    }
```

¿Tiene caracteres válidos?

Al igual que el caso con palabras reservadas, este **test case** prueba la función `hasValidChars` de la clase `Utilities.java`, que se asegura de que un nombre contega sólo caracteres válidos.

```
 @Test
    public void testHasValidChars_AllValid() {
        // Todas las letras son válidas
        assertTrue(Utilities.hasValidChars("abcABC123"));
        assertTrue(Utilities.hasValidChars("SELECT"));
        assertTrue(Utilities.hasValidChars("INSERT"));
        assertTrue(Utilities.hasValidChars("EMPLOYEES"));
    }
```

NO tiene caracteres válidos

Por otro lado, tenemos un caso contrario al **test case** anterior, donde aquí tenemos *nombres* que **NO** contiene caracteres válidos.

```
@Test
    public void tesHasValidChars_NotValid() {
        assertFalse(Utilities.hasValidChars("/home/path"));
        assertFalse(Utilities.hasValidChars("||name||"));
        assertFalse(Utilities.hasValidChars(".exec"));
    }
```

ParseQuery

Aquí tenemos un `test case` que prueba la función `parseQuery` de la clase `Parser.java`, que se encarga de recibir múltiples sentencias SQL y retornar si tienen una forma conocida o no.

```
```

```
@Test
    public void testParse() throws Exception {
        String[] query = { "masmcamd mwd cmqmd", "esto no es una
sentencia", "esto tampoco",
                "tengo hambre", "sexto cuatrimestr"};

        for (String s : query)
            assertEquals("No se reconoció la sentencia",
Parser.parseQuery(s));
    }
```

¿Existe un directorio?

En el siguiente **test case** probamos nuevamente la función `parseQuery`, pero enviándole una sentencia `USE` con una ruta que no existe, con el fin de verificar si efectivamente retorna que no existe el path.

```
@Test
    public void testParse() throws Exception {
        String[] query = { "masmcamd mwd cmqmd", "esto no es una
sentencia", "esto tampoco",
                "tengo hambre", "sexto cuatrimestr"};

        for (String s : query)
            assertEquals("No se reconoció la sentencia",
Parser.parseQuery(s));
    }
```

Problemas de permisos

En este **test case**, tenemos un intento de acceder a una base de datos en el *root* de linux, donde como sabemos se encuentran múltiples archivos sensibles y, por tanto, no deberíamos tener permisos para hacer cambios en dicho directorio.

```
@Test
    public void testUseDatabase_invalid2() throws Exception {
        FileManagement.initialValidations();
        String query = "USE /";

        Exception generatedException =
assertThrows(java.lang.Exception.class, () -> {
            Parser.parseQuery(query);
        });

        assertEquals("No tengo permisos para acceder a este directorio",
generatedException.getMessage());
    }
```

¿Hay path asignado?

En este **test case**, tenemos un intento de crear una tabla, sin embargo, tenemos que tener en cuenta que no se ha asignado un path de base de datos a usar, por lo cual no deberíamos poder crear nada.

```
@Test
    public void testCreateTable() throws Exception {
        FileManagement.initialValidations();
        FileManagement.setDatabasePath(null);
        String query = "CREATE TABLE EMP (ID int not null);";

        Exception generatedException =
assertThrows(java.lang.Exception.class, () -> {
            Parser.parseQuery(query);
        });

        assertEquals("No hay path asignado",
generatedException.getMessage());
    }
```

Paréntesis válidos

Muchas veces, al anidar múltiples funciones, puede ser que haya paréntesis que se queden sin cerrar, o que, en su defecto, haya paréntesis de cierre de más. Este **test case** se encarga de verificar que cada *paréntesis de apertura* tenga su respectivo *paréntesis de cierre*.

```
@Test
    public void testParenthesis() {
        FileManagement.initialValidations();
        FileManagement.setDatabasePath(new File("").getAbsolutePath() +
"/");
        String[] strs = { "SELECT * FROM (Locations, CREATE DATABASE(", "
((()))", "()()()(" };

        for (String s : strs)
            assertFalse(Parser.parenthesisCheck(s));
    }
```

¿Hay Primary Key?

Es necesario, por integridad de datos, que cada tabla tenga por lo menos una *primary key*. En el siguiente **test case** realizamos la prueba de qué pasaría al intentar crear una tabla sin asignar una PK.

```
@Test
    public void testCreateTable_notValidPrimaryKey() throws Exception {
        FileManagement.initialValidations();
        String path = new File("").getAbsolutePath() + "/";
```

```
        FileManagement.setDatabasePath(path);
        String query = "CREATE TABLE JOSHUA (ID int not null, NAME
varchar(50) not null, MONEY int not null, HEIGHT float not null)";


        Exception generatedException = assertThrows(Exception.class, () ->
{
            Parser.parseQuery(query);
        });

        assertEquals("No se puede crear una tabla sin PK",
generatedException.getMessage());
    }
```

Palabras reservadas en CREATE TABLE

La consulta `CREATE TABLE` espera, por cada columna asignada, al menos un nombre y un tipo de dato. Sin embargo, muchas veces el usuario puede intentar ingresar alguna palabra no válida dentro de la misma. Este **test case** se basa en verificar que sea válida la creación de tablas.

```
@Test
    public void testCreateTableInvalidArguments() throws Exception {
        FileManagement.initialValidations();
        String path = new File("").getAbsolutePath() + "/";
        FileManagement.setDatabasePath(path);
        String query = "create table add(id int not null primary key pedro
elizondo);";

        Exception generatedException = assertThrows(Exception.class, () ->
{
            Parser.parseQuery(query);
        });

        assertEquals("El nombre no puede ser una palabra reservada",
generatedException.getMessage());
        new File(new File("").getAbsolutePath() + "/JOSHUA.csv").delete();
        new File(new File("").getAbsolutePath() +
"/JOSHUA_aux.txt").delete();
    }
```

Argumento inválido dentro de select

Puede suceder que un usuario intente seleccionar una columna que **no existe**. Este **test case** prueba que todas las columnas que el usuario intente seleccionar existan efectivamente en la tabla.

```
@Test
    public void testArguments(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
```

```
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("select id, a from test");
        });
        assertEquals("Error en la sentencia: a", e.getMessage());
    }
```

Tabla no existe

Puede suceder que un usuario intente realizar alguna operación con una tabla que **no existe**. Este **test case** comprueba que se realiza la verificación que la tabla existe previo a realizar cualquier operación con ella.

```
@Test
    public void testTableDoesntExist(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("select * from a");
        });
        assertEquals("No se encontró la tabla: a", e.getMessage());
    }
```

Asignar dos PK

En mi sistema, no se puede asignar dos **PRIMARY KEY** a la vez a dos columnas distintas, y este **test case** se encarga de verificar eso.

```
@Test
    public void testCreateTableWithDuplicateKey(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("CREATE TABLE PRUEBA(ID INT NOT NULL PRIMARY
KEY, ID INT NOT NULL PRIMARY KEY)");
        });
        assertEquals("Nombre de columna repetido", e.getMessage());
    }
```

Update inválido

Al intentar realizar una consulta `UPDATE` puede suceder que la columna a modificar *no existe*. Este **test case** se basa en verificar eso.

```
@Test
    public void testUpdateInvalid(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
```

```
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("update test set a=2, b=3 where i = 1");
        });

        assertEquals(e.getMessage(), "Columna no encontrada en la tabla:
a");
    }
```

Insert into no coincide

Mi base de datos sólo soporta la consulta `INSERT INTO` explícita, que específica las columnas y los valores a insertar, por eso mismo, si los valores a insertar no coinciden de cualquier manera, entonces se debería lanzar una *excepción*.

```
@Test
    public void testInsertInvalid(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("insert into test (id, name) values (1,
'joshua', 2)");
        });

        assertEquals("Los valores a insertar y las columnas no coinciden",
e.getMessage());
    }
```

Integridad de PK en insert

La *primary key* no puede contener repetidos, esto implica que, al ingresar un nuevo dato, no se debe, de ninguna forma, ingresar una PK repetida. Este **test case** se encarga de verificar eso.

```
@Test
    public void testInsertRepeatedPrimaryKey(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("insert into test (id, name) values
(1,'jose')");
        });

        assertEquals("La primary key se repetiría, lo que compromete la
integridad de la base de datos", e.getMessage());
    }
```

Integridad de tipos en insert

La claúsula `INSERT INTO` también tiene en cuenta los tipos de datos, pues no se puede ingresar un valor de tipo `INT` en una columna de tipo `VARCHAR`. Este **test case** se encarga de verificar que la comprobación funcione correctamente.

```
@Test
    public void testInsertInvalidTypes(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("INSERT INTO test (id,name) values (15,34)");
            Parser.parseQuery("DELETE FROM test WHERE values id = 15");
        });

        assertEquals("Faltan comillas simples en 34", e.getMessage());
    }
```

Integridad de nulos en insert

La claúsula `INSERT INTO` también tiene en cuenta que podrían existir valores que, al momento de la creación de tabla, se hayan definido como `NOT NULL`. Este **test case** se encarga de verificar que no se le asigne un valor nulo a una tabla con el **constraint** `NOT NULL`.

```
@Test
    public void testNullVerification(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("INSERT INTO test (id,height) values
(15,1.20)");
        });

        assertEquals("Hay valores nulos que no deberían de serlo",
e.getMessage());
    }
```

Integridad de precisión

Puede suceder que, al momento de ingresar algún dato, no se respete la precisión establecida para dicho dato, en ese caso, entonces se lanzará una excepción.

```
@Test
    public void testPrecision(){
        Exception e = assertThrows(Exception.class, () -> {
            Parser.parseQuery("use /home/jarrazola/Documents/iti-271215-
poo-practica-2-JNArrazola/test/");
            Parser.parseQuery("INSERT INTO test (id,name,height) values
(15,'pedro',1.2314)");
```

```
        });

        assertEquals("Precisión equivocada", e.getMessage());
    }
```