



Catedráticos: Ing. Bayron López, Ing. Erick Navarro e Ing. Edgar Saban

Tutores académicos: Luis Lizama, Rainman Sián, Julio Arango.

CQL – Teacher

Primer Proyecto de Laboratorio

Contenido

1	Objetivos	6
1.1	Objetivo general	6
1.2	Objetivos específicos	6
2	Descripción	6
2.1.1	Lenguaje Unificado de Paquetes (LUP)	6
2.1.2	Compi Query Lenguaje (CQL)	6
2.1.3	Chison	7
2.1.4	CQL-Client.....	7
2.1.5	Login.....	7
2.1.6	Panel de información	8
2.1.7	Salida de datos y Consola	8
2.1.8	Modos de edición	9
2.1.9	Servidor de Bases de datos.....	12
2.2	Flujo del Proyecto	13
2.2.1	Flujo General de la aplicación	13
2.2.2	Flujo Específico de la aplicación.....	14
2.3	Notación utilizada para la definición de los lenguajes	15
2.3.1	Sintaxis y ejemplos	15
2.3.2	Asignación de colores.....	15
2.3.3	Expresiones.....	15

2.3.4	Cerradura positiva	15
2.3.5	Cerradura de Kleene	15
2.3.6	Opcionalidad.....	16
2.4	Definiciones generales de CQL.....	16
2.4.1	Identificadores	16
2.4.2	Case Insensitive	16
2.4.3	Comentarios	16
2.4.4	Valor nulo	17
2.5	Tipos de dato	17
2.5.1	Tipos primitivos.....	17
2.5.2	Secuencias de escape.....	18
2.5.3	User Types	19
2.5.4	Casteos primitivos en CQL	21
3	Lenguaje de Definición de Datos (DDL).....	23
3.1	Crear Base de datos	23
3.2	Sentencia Use.....	23
3.3	Sentencia Drop	23
3.4	Crear Tabla	24
3.4.1	Tipo Dato Counter	24
3.4.2	Llaves Primarias	24
3.4.3	Tipos de datos para definición de columnas.....	24
3.5	Alter Table.....	26
3.6	Drop Table	26
3.7	Truncate.....	27
4	Lenguaje de Control de Transacciones (TCL).	27
4.1	Commit.....	27
4.2	Rollback	27
5	Lenguaje de Control de Datos (DCL).	28
5.1	Crear Usuario.....	28
5.2	Grant.....	28
5.3	Revoke.....	28
6	Lenguaje de Manipulación de datos (DML).....	29

6.1	Insertar	29
6.2	Actualizar	30
6.3	Eliminar	30
6.4	Seleccionar	31
6.5	Batch.....	32
6.6	Funciones de Agregación	33
6.7	Clausula Where.....	34
6.8	Uso de colecciones y objetos como componentes de una tabla	35
6.8.1	Maps.....	35
6.8.2	Sets	36
6.8.3	Listas	37
6.8.4	User Types	38
7	Lenguaje de Control de Flujo (FCL)	39
7.1.1	Paradigma de programación	39
7.2	Sintaxis	39
7.2.1	Bloque de sentencias	39
7.2.2	Signos de agrupación	39
7.2.3	Variables.....	40
7.2.4	Declaración de variables	40
7.2.5	Asignación de variables.....	40
7.2.6	Valores predeterminados para las variables.....	41
7.2.7	Operadores aritméticos	41
7.2.8	Operadores relacionales.....	46
7.2.9	Operaciones lógicas	47
7.2.10	Operador ternario	48
7.2.11	Sentencias de Selección	49
7.2.12	Sentencias cíclicas o bucles.....	51
7.2.13	Collections.....	52
7.2.14	Funciones.....	56
7.2.15	Procedimientos.....	57
7.2.16	Funciones nativas sobre cadenas	60
7.2.17	Funciones nativas de abstracción	62

7.2.18	Sentencias de transferencia	64
7.2.19	Cursores.....	66
7.2.20	Función LOG	68
7.2.21	Sentencia throw.....	69
7.2.22	Sentencia try Catch	69
8	Lenguaje Unificado de Paquetes (LUP)	70
8.1	Solicitudes del cliente al servidor:	70
8.1.1	Paquete de Inicio de Sesión	70
8.1.2	Paquete de Fin de Sesión	70
8.1.3	Paquete de Consulta:	71
8.2	Respuestas del servidor al cliente:.....	71
8.2.1	Paquete de Datos.....	71
8.2.2	Paquete de Error	72
8.2.3	Paquete de Mensaje.....	72
8.2.4	Paquete de Estructura	72
9	El Lenguaje de Almacenamiento de Datos (CHISON)	74
9.1	Notación	74
9.1.1	Datos Primitivos.....	74
9.2	Definición de los archivos	75
9.2.1	Archivo Principal.....	75
9.3	Importar Archivo.....	82
10	Excepciones.....	82
11	Apéndice A: Precedencia y asociatividad de operadores	84
12	Manejo de errores	85
12.1	Tipos de errores	86
12.2	Contenido mínimo de tabla de errores	86
12.3	Método de recuperación	86
13	Archivos de Entrada.....	86
14	Entregables y Restricciones.....	87
14.1	Entregables.....	87
14.2	Restricciones.....	87
14.3	Requisitos mínimos.....	87

14.4 Entrega del proyecto	90
---------------------------------	----

Contenido Ilustraciones

Ilustración 1: Ejemplo Login	7
Ilustración 2: Panel de información	8
Ilustración 3: Consola de salida	8
Ilustración 4: Tablas de salida	8
Ilustración 5: Modulo de edición	9
Ilustración 6: Modo principiante	9
Ilustración 7: Modo intermedio	11
Ilustración 8: Modo avanzado	12
Ilustración 9: Cliente - Servidor	13
Ilustración 10: Flujo general	13
Ilustración 11: Flujo específico	14

1 Objetivos

1.1 Objetivo general

Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 2 en el desarrollo de una aplicación.

1.2 Objetivos específicos

- Utilizar herramientas para la generación de analizadores léxicos y sintácticos.
- Realizar análisis semántico a los diferentes lenguajes a implementar.
- Que el estudiante aplique los conocimientos adquiridos durante la carrera para el desarrollo de un intérprete.
- Que el estudiante sea capaz de implementar los conocimientos adquiridos en el curso tecnologías modernas.

2 Descripción

CQL-Teacher es una aplicación cliente–servidor, que permitirá interactuar con los usuarios para que puedan familiarizarse con el uso y beneficios de una base de datos, de una forma bastante fácil e intuitiva, proveerá bastantes modos para que así puedan trabajar de acuerdo con su nivel de conocimiento. Además, el sistema de base de datos también contará con instrucciones y elementos basados de un lenguaje estructurado, tales como variables, ciclos, sentencias condicionales y tipos de datos definidos por el usuario, CQL-Teacher está compuesto principalmente por:

2.1.1 Lenguaje Unificado de Paquetes (LUP)

LUP es un lenguaje de paquetes basado en XML que permitirá la comunicación entre el cliente y el servidor de bases de datos. Contará con una serie de paquetes tales como consulta, login, error, datos, notificación, entre otros. Esto le permitirán al cliente determinar la manera que debe mostrarse la información.

2.1.2 Compi Query Lenguaje (CQL)

Es un lenguaje de consultas que permite la manipulación de la información almacenada en el servidor de bases de datos. Este lenguaje, aunque trabaja como uno solo, se divide en 5 grandes grupos, los cuales son:

- **Lenguaje de Definición de Datos (DDL).** Permite llevar a cabo las tareas de definición de las estructuras que almacenarán los datos.
- **Lenguaje de Manipulación de datos (DML).** Permite la consulta, agregación, modificación o eliminación de los datos dentro de las estructuras.
- **Lenguaje de Control de Datos (DCL).** Se encarga de la administración de usuarios y permisos.
- **Lenguaje de Control de Transacciones (TCL).** Garantiza la integridad de la información por medio de las sentencias commit y rollback.

- **Lenguaje de Control de Flujo (FCL)** que agrupa todas las sentencias de transferencia, selección y repetición, así como el manejo de variables, estructuras, entre otros.

2.1.3 Chison

Chison es un lenguaje basado en JSON, a través del cual el servidor de bases de datos garantizará la persistencia de la información. Los archivos en formato chison se actualizarán cada vez que se ejecute una sentencia *commit* o el servidor se apague. Se requiere la implementación de un intérprete para este lenguaje que le permita al servidor recuperar los datos almacenados hasta antes que se detuvo, en el caso de una sentencia *rollback* le permitirá recuperarse hasta un último punto de rescate, el ultimo *commit*.

2.1.4 CQL-Client

La aplicación cliente, CQL-Client, es una aplicación web que permitirá al usuario interactuar con el servidor de bases de datos, esta contará con una serie de componentes que le brindan al usuario una experiencia agradable al momento de utilizarlo. Tiene 3 distintos modos de dificultad para la edición de código de alto nivel, así como un login para garantizar la seguridad de la información.

2.1.5 Login

CQL-Client cuenta con un mecanismo de autenticación para garantizar la seguridad de la base de datos, únicamente los usuarios logueados podrán hacer uso del IDE. Los usuarios serán almacenados en la base de datos. Por defecto, el sistema debe contar con el usuario admin. Las credenciales para este usuario son: admin y password: admin. Con este usuario se podrán crear más usuarios.

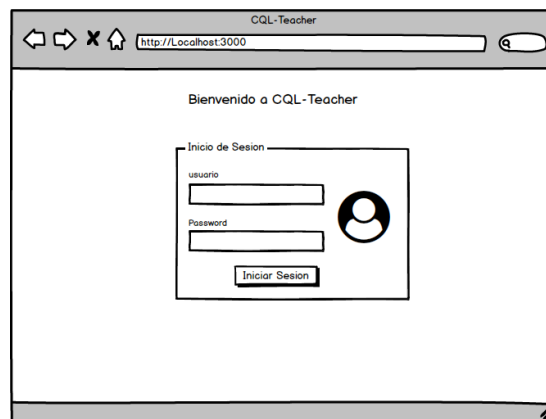


Ilustración 1: Ejemplo Login

2.1.6 Panel de información

Le permite al usuario visualizar un menú en forma de árbol en el que se deben de mostrar en todo momento las bases de datos existentes y todos los elementos que la conforman, tales como: tablas, objetos, procedimientos almacenados y todo lo demás que el estudiante considere necesario. Únicamente se mostrarán los datos a los que el usuario tenga acceso.

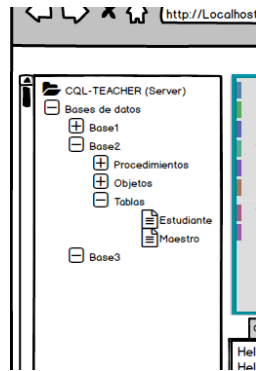


Ilustración 2: Panel de información

2.1.7 Salida de datos y Consola

Para cada consulta de tipo *Select* realizada a la base de datos, se debe de crear una nueva pestaña que muestre en forma de tabla los datos obtenidos (ver ilustración 4), para todos los demás mensajes que se deban realizar, como impresiones y reportes de error, se deben de visualizar en consola (ver ilustración 3). La consola no es más que una pestaña independiente que permite visualizar las salidas de texto plano.

Ejemplo de salidas en Consola

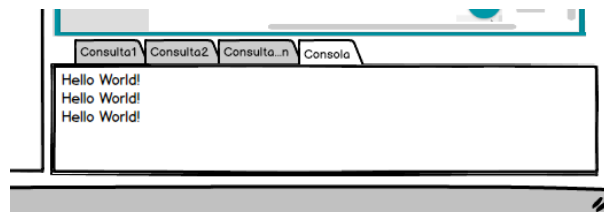


Ilustración 3: Consola de salida

Ejemplo de resultados de una consulta tipo *Select*

A screenshot of a web browser window showing a tab labeled 'Consola'. The console displays a table with three columns: 'Id', 'Nombre', and 'Apellido'. The table has three rows of data. The first row is highlighted in blue. The browser's address bar shows 'http://localhost'.

Ilustración 4: Tablas de salida

2.1.8 Modos de edición

Al iniciar sesión en CQL-Client, este desplegará una pantalla en el cual es posible seleccionar el modo deseado, la aplicación contará con 3 modos distintos: principiante, intermedio y avanzado.



Ilustración 5: Módulo de edición

2.1.8.1 Modo Principiante

El modo principiante contará con un sistema de bloques de código CQL que podrán arrastrarse hacia el área de edición por medio de drag-and-drop. Este contará con una opción para poder eliminar los bloques que no estemos utilizando y también con un botón que nos permita ejecutar las consultas.

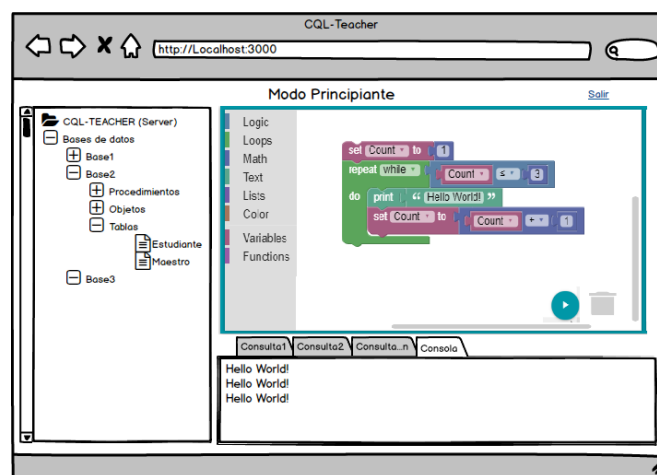


Ilustración 6: Modo principiante

En este modo de edición el usuario tiene acceso a las sentencias DML, estos bloques podrán ser apilados y/o conectados entre sí para formar un bloque de sentencias a ejecutar.

En este modo de edición las sentencias se limitan a

- Select
- Insert
- Update
- Delete

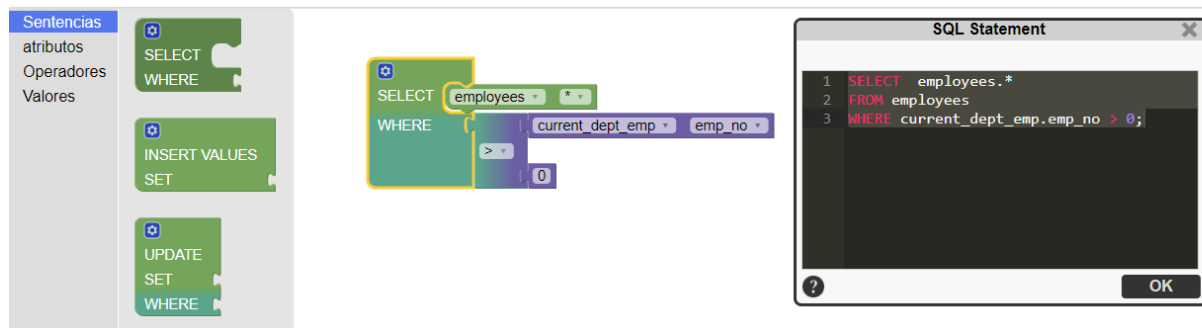


Ilustración 7: Sentencias en Modo Principiante

Bloques

- **Sentencias.** Contiene los bloques para las instrucciones Select, Insert, Update y Delete
- **Atributos.** Contiene bloques con listas para poder seleccionar nombre de Tablas y las columnas. Cuenta con el comodín "*" que permite seleccionar todos los campos para la instrucción Select
- **Operadores.** Contiene los bloques para poder agregar comparaciones booleanas, expresiones aritméticas y expresiones lógicas
- **Valores.** Contiene los bloques para poder agregar valores tales como enteros, decimales, valores booleanos, cadenas, fechas. También contiene el bloque para poder agregar lista de valores.

2.1.8.2 Modo Intermedio

El modo intermedio, además de incorporar una mayor cantidad de bloques, tales como sentencias de control, sentencias de transferencia, declaración de variables, entre otras; permite visualizar en un área contigua el editor de bloques, mostrando la construcción del código de alto nivel conforme se va armando la entrada. Al igual que el nivel principiante, contará con un botón para ejecutar la consulta y una opción para eliminar los bloques no utilizados.

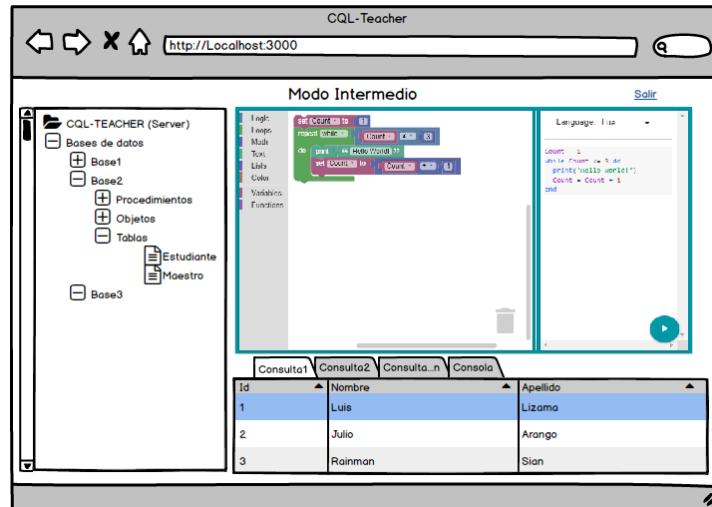


Ilustración 8: *Modo intermedio*

En este modo de edición las sentencias se limitan a

- Definición de variables
- Asignaciones
- If / Else
- Switch
- While
- Do While
- For
- Llamadas a funciones y procedimientos

Bloques:

- **Variables.** Permite la creación de variables con los tipos válidos de CQL (solo primitivos); también permite la asignación de variables con expresiones
- **Sentencias de control.** Contiene los bloques para poder agregar sentencias If / Else y Switch
- **Ciclos.** Contiene los bloques para poder agregar sentencias While, Do-While y For.
- **Operadores.** Contiene los bloques para poder agregar comparaciones booleanas, expresiones aritméticas, expresiones lógicas y llamadas a funciones.
- **Valores.** Contiene los bloques para poder agregar valores tales como enteros, decimales, booleanos, cadenas, fechas.
- **Procedimientos.** Contiene los bloques para poder llamar a procedimientos primitivos como LOG y las definidas por el usuario/

2.1.8.3 Modo Avanzado

El modo avanzado es un editor de texto sin restricciones desde el cual el usuario podrá ingresar cualquier sentencia CQL. Además de esto, se desea que el editor cuente múltiples pestañas y las opciones de guardar archivo, abrir archivo, ejecutar el script completamente o solo las instrucciones seleccionadas.

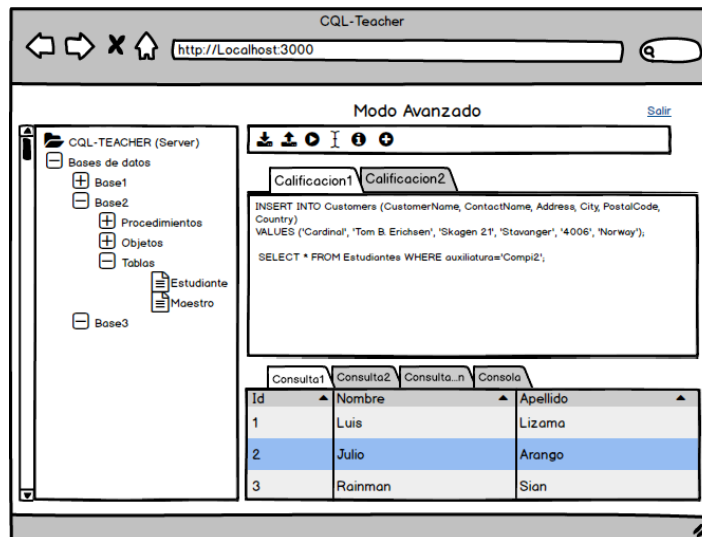


Ilustración 9: Modo avanzado

2.1.9 Servidor de Bases de datos

El servidor de base de datos será el encargado de manipular y almacenar la información por medio del lenguaje CQL. Utilizará una arquitectura de tipo REST, de esta manera podrá darles soporte a múltiples clientes al mismo tiempo (ver ilustración 9).

El envío y recepción de paquetes entre este servidor y el cliente será por medio del lenguaje LUP, por lo que el estudiante deberá implementar un intérprete de este lenguaje tanto en el servidor como en el cliente.

El almacenamiento de los datos se realizará a través de archivos en formato Chison, para garantizar la integridad de la información cada vez que se encuentre la sentencia *commit* o el servidor se apague, deberá de actualizar los archivos, mientras que cuando se encuentre la sentencia *rollback* se deberán retrotraer los cambios hechos dentro de la unidad hasta un punto de rescate (último commit realizado). Al momento de arrancar el servidor, se deberá implementar un intérprete del lenguaje Chison que le permita al servidor de bases de datos recuperar toda la información que tenía antes de apagarse.

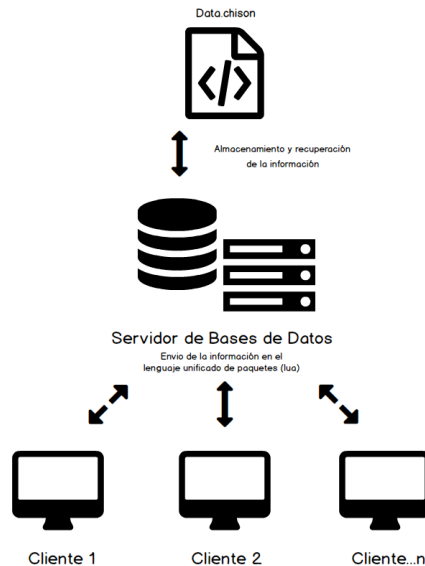


Ilustración 10: Cliente – Servidor

2.2 Flujo del Proyecto

CQL-Teacher es un sistema de gestión de bases de datos no relacional, orientado a objetos y especialmente diseñado para que todas las personas puedan iniciarse en el mundo de las bases de datos de una manera sencilla, divertida y sumamente intuitiva.

2.2.1 Flujo General de la aplicación

La solución propuesta consta de dos partes, un cliente desde el cual el usuario podrá realizar cualquier tipo de consultas y un servidor de bases de datos con arquitectura REST que será el encargado de manipular y almacenar toda la información. La comunicación entre cliente y servidor se realizará mediante un lenguaje de paquetes denominado LUP. El servidor de base de datos contará con el lenguaje Chison, cada vez que el servidor se detenga, este deberá de almacenar los datos en este formato para garantizar la integridad de la información. Se solicita que el estudiante implemente un intérprete del lenguaje Chison para poder recuperar los datos al momento de reiniciar el servidor. El flujo estándar de la aplicación se detalla en el siguiente diagrama:

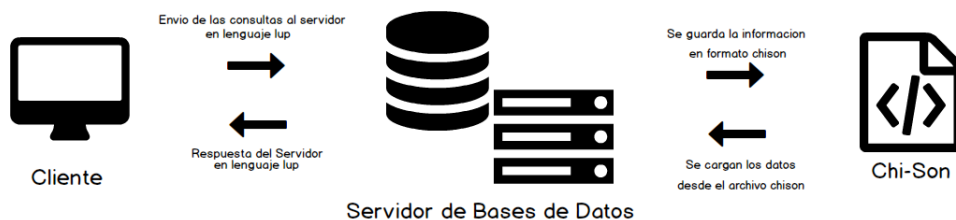


Ilustración 11: Flujo general

2.2.2 Flujo Específico de la aplicación

A continuación, se detalla el proceso específico a través del cual el usuario puede realizar una consulta.

1. El usuario ingresa a la plataforma, inicia sesión y selecciona el modo deseado.
2. El usuario ingresa la entrada deseada utilizando cualquiera de los modos (Principiante, Intermedio, Avanzado) y presiona el botón ejecutar.
3. El Cliente genera un paquete de consulta en lenguaje LUP y se lo envía al servidor de bases de datos.
4. El servidor interpreta el paquete de consulta, separa del paquete la entrada ingresada por el usuario y ejecuta las instrucciones.
5. El servidor genera paquetes de datos, notificación, error, etc. Según haya sido el resultado de las consultas.
6. Los datos modificados con las consultas son actualizados en los archivos en formato Chison para garantizar la integridad de los datos.
7. Se envían los paquetes de vuelta al cliente.
8. El cliente interpreta los paquetes y separa los datos deseados, si es un paquete de datos, lo muestra en formato de tabla, si fuera notificación o error, lo muestra en la pestaña de notificaciones.
9. Se termina la petición.

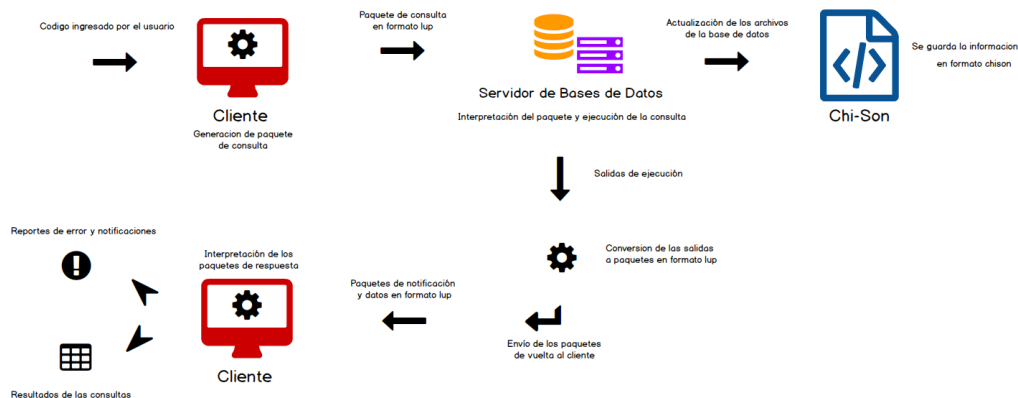


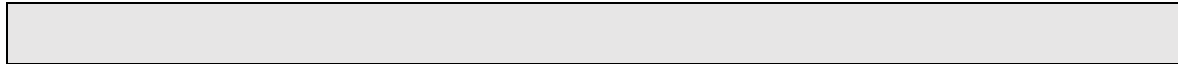
Ilustración 12: Flujo específico

2.3 Notación utilizada para la definición de los lenguajes

A continuación, se definirán todos los lenguajes descritos para CQL-Teacher, para una mayor comprensión, se utilizarán las siguientes notaciones cuando se hará referencia al código del lenguaje que se esté definiendo. Todo esto no es necesario aplicarlo en el editor del Cliente.

2.3.1 Sintaxis y ejemplos

Para definir la sintaxis y ejemplos de sentencias de código de alto nivel se utilizará un rectángulo gris.



2.3.2 Asignación de colores

Dentro del enunciado y en el editor de texto de la aplicación, para el código de alto nivel, seguirá el formato de colores establecido en la Tabla 1.

*Tabla 1: código de colores
Fuente: Elaboración propia.*

COLOR	TOKEN
AZUL	Palabras reservadas
NARANJA	Cadenas
MORADO	Números
GRIS	Comentario
CAFÉ	Salida por consola
NEGRO	Otro

2.3.3 Expresiones

Cuando se haga referencia a una 'expresión', se hará referencia a cualquier sentencia que devuelve un valor. Por ejemplo:

- Una operación aritmética, relacional o lógica,
- Un acceso a un variable u objeto.
- Una llamada a una función

2.3.4 Cerradura positiva

Esta será usada cuando se desee definir que un elemento del lenguaje podrá venir una o más veces: (elemento)+

2.3.5 Cerradura de Kleene

Esta será usada cuando se desee definir que un elemento del lenguaje podrá venir cero o más veces: (elemento)*

2.3.6 Opcionalidad

Esta será usada cuando se desee definir que un elemento del lenguaje podrá o no venir (cero o una vez): (elemento)?

2.4 Definiciones generales de CQL

CQL contiene distintos lenguajes para distintas tareas, aunque todos estos pueden llegar a tener una relación directa para la ejecución de tareas, se definirán por aparte para mayor comprensión del estudiante, a continuación, se describen las características generales que compartirán todos los lenguajes que se definirán más adelante.

2.4.1 Identificadores

Un identificador será utilizado para dar un nombre a variables, métodos o estructuras. Un identificador es una secuencia de caracteres alfabéticos [A-Z a-z] incluyendo el guion bajo [_] o dígitos [0-9] que comienzan con un carácter alfabético o guion bajo.

Ejemplos de identificadores validos

```
Este_es_un_id_valido_1
_es_un_id_valido_1
Este_un_id_valido_1
```

Ejemplos de identificadores no validos

```
0id
ls-15
```

2.4.2 Case Insensitive

Todos los lenguajes definidos para CQL serán case insensitive, esto quiere decir que no diferenciara mayúsculas con minúsculas, por lo tanto, el identificador var hace referencia al mismo identificador Var, igual para palabras reservadas.

2.4.3 Comentarios

Un comentario es un componente léxico del lenguaje que no es tomado en cuenta para el análisis sintáctico de la entrada.

Existirán dos tipos de comentarios:

- Los comentarios de una línea que serán delimitados al inicio con los símbolos “//” y al final con un carácter de finalización de línea
- Los comentarios con múltiples líneas que empezarán con los símbolos “/*” y terminarán con los símbolos “*/”.


```
//este es un ejemplo de un lenguaje de una sola línea  
/*  
este es un ejemplo de un lenguaje de múltiples líneas.  
*/
```

2.4.4 Valor nulo

Se reservará la palabra “**null**” en el lenguaje CQL y está se utilizará para hacer referencia a la nada, indicará la ausencia de valor

2.5 Tipos de dato

Todos los lenguajes serán de tipado estático, esto quiere decir que al momento de compilación ya se conoce el tipo de cada variable y de cada expresión. También es un lenguaje de programación con tipado fuerte ya que el tipo de una variable define el rango de valores que esta puede tomar.

2.5.1 Tipos primitivos

Se utilizarán los siguientes tipos de dato:

- **Entero:** este tipo de dato aceptará valores numéricos enteros, será identificado con la palabra reservada “int”.
- **Decimal:** este tipo de dato aceptará números de coma flotante de doble precisión¹, será identificado con la palabra reservada “double”.
- **Cadena:** este tipo de dato aceptará cadenas de caracteres, que deberán ser encerradas dentro de comillas dobles (“**caracteres**”), será identificado con la palabra reservada “string”.
- **Booleano:** este tipo de dato aceptará valores de verdadero y falso que serán representados con palabras reservadas que serán sus respectivos nombres (**true**, **false**), será identificado con la palabra reservada “boolean”.
- **Date:** este tipo de dato será el encargado de almacenar fechas, son almacenadas como en formato ‘**yyyy-mm-dd**’ encerradas entre comillas simples, siendo su orden (año, mes y día), será identificado con la palabra reservada “date”
- **Time:** este tipo de dato será el encargado de almacenar el tiempo son almacenados como en formato ‘**hh:mm:ss**’ encerradas entre comillas simples, siendo su orden (horas, minutos y segundos) en formato 24 horas

¹ El formato en coma flotante de doble precisión es un formato de número de computador u ordenador que ocupa 64 bits en su memoria y representa un amplio y dinámico rango de valores mediante el uso de la coma flotante. Este formato suele ser conocido como binary64 tal como se especifica en el estándar IEEE 754.

Tabla 2: rangos y requisitos de almacenamiento de cada tipo de dato
Fuente: Elaboración propia.

TIPO DE DATO	RANGO	TAMAÑO
INT	<code>[-2147483648, 2.147.483.647]</code>	4 bytes
DOUBLE	<code>[-9223372036854775800, 9223372036854775800]</code>	8 bytes
BOOLEAN	<code>true, false</code>	1 byte
STRING	<code>[0, 2.147.483.647]</code> caracteres	Variable
DATE	<code>[0000-2999]-[00-12]-[00-31]</code>	Variable
TIME	<code>[00-24]:[00-60]:[00-60]</code>	Variable

Consideraciones

- Cuando se efectúen operaciones con datos primitivos y el resultado de dicha operación sobrepase el rango establecido se deberá mostrar un error en tiempo de ejecución.

2.5.2 Secuencias de escape

Las secuencias de escape se utilizan para definir ciertos caracteres especiales dentro de cadenas de texto. Las secuencias de escape disponibles son las siguientes:

Secuencia de escape	Descripción
<code>\'</code>	Comillas simples
<code>\"</code>	Comillas dobles
<code>\?</code>	Signo de interrogación
<code>\\</code>	Barra invertida
<code>\n</code>	Nueva línea
<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulación horizontal

2.5.3 User Types

CQL le proveerá al usuario la capacidad de crear sus propios tipos de datos, estos funcionarán como estructuras de diferentes tipos de datos y existirán de forma global en todos los ámbitos, estos tipos de datos también podrán ser modificados y eliminados.

Sintaxis para la creación

```
Create type [If not exists]? <Nombre>(  
    <Nombre> <TipoDato>  
    (,<Nombre><TipoDato>)*  
);
```

Esto únicamente creará la definición del objeto mas no se podrá utilizar, para instanciar un objeto se utilizará la siguiente sintaxis:

Sintaxis

```
User_Type nombre ; //Crea la declaración del objeto  
User_Type nombre = new User_Type; // Crea una instancia del objeto
```

Se pueden hacer instancias de User Types con valores dados, para ello se pone entre llaves todos los valores que debe tomar el objeto, separados por coma, se toma de forma relativa, por lo cual el primer valor corresponde a la primera variable del user type, se debe verificar que venga exactamente todos los elementos del user type y todos del tipo que les corresponde.

Sintaxis

```
User_Type nombre = {<LISTA_VALORES>; // Crea una instancia del  
objeto
```

Ejemplo

```
Create type Estudiante(  
    carnet int,  
    String nombre  
);  
Estudiante @est; //Crea la declaración del objeto  
@est = new Estudiante; // Crea una instancia del objeto  
Estudiante @est2 = new Estudiante; // Crea otra instancia del objeto  
Estudiante @est3 = {201504481, "Julio Arango"} // Crea otra instancia del objeto
```

Consideraciones

- El tipo de dato de los elementos del User Type, pueden ser de tipo primitivo, [Collections](#) e incluso otros user types.

2.5.3.1 Acceso a los elementos de user types

Para acceder a los valores de los elementos de los user types se utiliza el nombre de la instancia del objeto, seguido del nombre del elemento al cual se quiere acceder, se puede asignar y obtener valor de esta forma.

Sintaxis

```
<VARIABLE> '.' <NOMBRE_ELEMENTO>
```

Ejemplo

```
Estudiante @est3 = {201504481, "Julio Arango"} // Crea otra instancia del objeto  
Int @carnet = @est3.carnet;  
@est3.nombre = "Julius"
```

2.5.3.2 Alterar User types

Para la modificación se pueden agregar o eliminar datos, para agregar se escoge la opción de Add y agrega todos los elementos dentro del Alter, para eliminar datos se escoge la opción de Delete y elimina todos los datos que tengan los nombres dentro del Alter.

Sintaxis

```
// Agregar  
Alter type <Nombre> Add(  
    <Nombre> < CQL_TIPO >  
    (,<Nombre> < CQL_TIPO >)*  
);  
// Eliminar  
Alter type <Nombre> Delete(  
    <Nombre>  
    (,<Nombre>)*  
);
```

Ejemplo

```
Alter type Estudiante Add(  
    Int CUI,  
    String direccion;  
);
```

```
Alter type Estudiante Delete(  
    Carnet,  
    direccion  
);
```

2.5.3.3 Eliminar User types

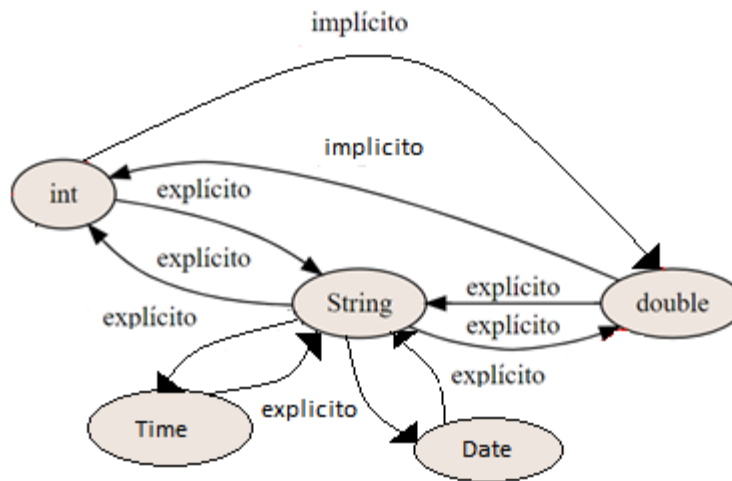
Elimina el tipo de dato creado por el usuario

Sintaxis

```
Delete type <Nombre>;
```

2.5.4 Casteos primitivos en CQL

Figura 1: casteos permitidos
Fuente: Elaboración propia.



2.5.4.1 Casteo implícito

Este tipo de casteo (también llamado conversión ancha) es una conversación entre dos tipos sin necesidad de una conversión directa del valor, únicamente se realiza un cambio de tipo.

Las únicas conversiones implícitas que soportará CQL son:

- **Int – Double:** No se realiza ninguna acción sobre el valor.
- **Double – Int:** Se realiza a trincar el valor del entero.

Ejemplos

```
Int a = 10;  
Double b = a; /// valor de b = 10.0  
  
Double c = 2.25;  
Int d = c; // valor de d = 2
```

2.5.4.2 Casteo explícito

Este tipo de casteo (también llamado conversión estrecha) es una conversión entre dos tipos, para ello es primero determinar a qué tipo de dato se desea castear.

Sintaxis

```
(<TIPO_CASTEO>) Expresión
```

Consideraciones:

- Aplica para tipos de casteo a Time, Date y String.
- Si el valor de la expresión no puede ser contenido en el tipo de casteo debe mostrarse un error en tiempo de ejecución.
- Para las conversiones de String a Date y Time, la cadena debe de estar en formato de la conversión.

3 Lenguaje de Definición de Datos (DDL).

DDL es el lenguaje de programación propio de CQL que permite llevar a cabo las instrucciones para la definición de las estructuras que almacenarán los datos, como la creación de bases de datos y tablas.

3.1 Crear Base de datos

Una de las sentencias principales de DDL es la creación de base de datos, esta sentencia recibirá de forma obligatoria el nombre de la base de datos.

Sintaxis

```
CREATE DATABASE [IF NOT EXISTS]? <NOMBRE_BD>;
```

Consideraciones

- Si se intenta crear una base de datos que ya existe se debe desplegar un error en consola a menos que la directiva **IF NOT EXISTS** se haya indicado.

3.2 Sentencia Use

Esta sentencia es la encargada de determinar qué base de datos es la que se utilizará para la ejecución de todas las siguientes instrucciones.

Sintaxis

```
USE <NOMBRE_BD> ;
```

Consideraciones

- En caso no exista la base de datos que se intenta utilizar, se debe desplegar un error en consola y no se utilizará ninguna otra base de datos.

3.3 Sentencia Drop

Esta sentencia será la encargada de eliminar una base de datos, se eliminan todos sus datos, procedures, funciones y todo lo relacionado a esta base de datos.

Sintaxis

```
DROP DATABASE <NOMBRE_BD>;
```

Consideraciones

- En caso no exista la base de datos que se intenta eliminar, se debe desplegar un error en consola y no se eliminará ninguna otra base de datos.

3.4 Crear Tabla

Para crear una nueva tabla se usa la sentencia **CREATE TABLE**. Una tabla en CQL tiene un nombre y una lista de columnas, debe existir al menos una columna y opcionalmente una definición de clave primaria.

3.4.1 Tipo Dato Counter

El tipo counter es un tipo de dato especial para la definición de columnas. Una columna de tipo counter tiene un valor de tipo INT auto-incrementable. Este valor no puede ser actualizado, se define con la palabra reservada “**counter**”.

Consideraciones

- Solo puede ser usado por columnas definidas como parte de una **PRIMARY KEY**
- Si la primary key es compuesta, todas las columnas deben ser definidas como counter o ninguna de ellas.
- Este valor no puede ser actualizado por la sentencia **UPDATE**
- Una columna de este tipo de dato puede ser definida en la creación de tablas y también al momento de agregar columnas.

3.4.2 Llaves Primarias

Una clave primaria es un campo o grupo de campos que identifica en forma única un registro. Ningún otro registro puede tener la misma clave primaria, si se almacenan dos valores iguales que son llaves primarias, el registro no se insertara y se mostrara un error de semantica.

3.4.3 Tipos de datos para definición de columnas

Para definir las columnas de una tabla, es necesario especificar el tipo que tendrá la columna, si se inserta cualquier valor que no sea válido con este tipo se deberá de descartar la inserción y reportar un error en tiempo de ejecución. CQL es un lenguaje totalmente orientado a objetos, por lo que es permitido el almacenamiento de objetos y colecciones dentro de las tablas.

Las columnas podrán ser de cualquiera de los siguientes tipos:

- String
- Int
- Double
- Date

- Time
- Boolean
- Counter
- Cualquier tipo definido por el usuario. (User Types)
- Cualquier [Collections](#) del lenguaje. (Map,Set,List)

Consideraciones

- Si se intenta crear una tabla que ya existe se debe desplegar un error en consola a menos que la directiva **IF NOT EXISTS** se haya indicado.
- Una columna se compone de un nombre y su tipo, el cual restringe los valores que esta columna puede almacenar. Adicionalmente, cada definición de columna puede tener el modificador **PRIMARY KEY**
- Si se desea tener llaves primarias compuestas se realiza una definición por aparte.
- La llave primaria es única y no puede ser null.

Sintaxis crear tabla

```
CREATE TABLE [ IF NOT EXISTS ]? <NOMBRE_TABLA>
'('
    column_definition
    ( ',' column_definition ) *
    [ ',' PRIMARY KEY '(' primary_key ')' ]
')';
```

Sintaxis crear columna

```
<NOMBRE_COLUMNA> <CQL_TIPO> (PRIMARY KEY)?
```

Sintaxis llave primaria compuesta

```
PRIMARY KEY '(' <NOMBRE_COLUMNA> ( ',' <NOMBRE_COLUMNA> ) * ')'
```

Ejemplo

```
// Tabla con llave simple
CREATE TABLE EspeciesDeMono (
    especie string PRIMARY KEY,
    nombre_comun string,
    poblacion int,
```

```

        tamaño_promedio int
    );
    //Tabla con llave compuesta
    CREATE TABLE Muro (
        usuario_id int,
        mes_publicado int,
        hora_publicado double,
        contenido string,
        publicado_por string,
        PRIMARY KEY (usuario_id, mes_publicado, hora_publicado)
    );

```

3.5 Alter Table

Esta sentencia nos permite agregar o remover columnas de una tabla. Para agregar nuevas columnas se hace a través de la instrucción **ADD**. Las nuevas columnas agregadas no pueden ser parte de la PRIMARY KEY. Para remover columnas lo hacemos a través de la instrucción **DROP**.

Sintaxis agregar columna

```

ALTER TABLE <NOMBRE_TABLA>
ADD <NOMBRE_COLUMNA> <CQL_TIPO>
(, <NOMBRE_COLUMNA> <CQL_TIPO>)* ',';

```

Sintaxis eliminar columna

```

ALTER TABLE <NOMBRE_TABLA>
DROP <NOMBRE_COLUMNA>
(, <NOMBRE_COLUMNA>)* ',';

```

3.6 Drop Table

Para borrar una tabla se usa la sentencia **DROP TABLE**. Borrar una tabla es una sentencia irreversible. El resultado será la eliminación tanto de la tabla, así como de sus datos.

Sintaxis

```

DROP TABLE [ IF EXISTS ]? <NOMBRE_TABLA>;

```

Ejemplo

```

DROP TABLE IF EXISTS usuarios;

```

Consideraciones

- Si se intenta eliminar una tabla que ya existe se debe desplegar un error en consola a menos que la directiva **IF EXISTS** se haya indicado.

3.7 Truncate

Una tabla puede ser truncada usando la sentencia **TRUNCATE TABLE**. Truncar una tabla remueve permanentemente todos los datos de la table sin eliminar la tabla.

Sintaxis

```
TRUNCATE TABLE <NOMBRE_TABLA>;
```

4 Lenguaje de Control de Transacciones (TCL).

Es el lenguaje utilizado para manejar las transacciones en la base de datos. Son utilizados para manejar los cambios que se realizan a la información de la base de datos. Consta de 2 sentencias básicas:

4.1 Commit

Es la sentencia almacenará los cambios de manera permanente en los archivos CHISON. Cada vez que se encuentre esta sentencia, los archivos se deberán sobrescribir con la información más reciente.

Sintaxis

```
COMMIT ;
```

4.2 Rollback

Esta sentencia causara que todos los cambios en la información desde la última sentencia commit sean descartados por la base de datos.

Sintaxis

```
ROLLBACK ;
```

5 Lenguaje de Control de Datos (DCL).

Es el lenguaje que se encarga de controlar el acceso a la información contenida en la base de datos. Los usuarios únicamente podrán acceder a las bases de datos sobre las que tiene permiso.

5.1 Crear Usuario

Esta sentencia permitirá agregar usuarios nuevos a la base de datos, por defecto, estos usuarios no tendrán ningún permiso más que a las bases de datos que ellos mismos creen.

Sintaxis

```
CREATE USER <NOMBRE_USUARIO> WITH PASSWORD <CONTRASEÑA>;
```

Ejemplo

```
CREATE USER Pedro WITH PASSWORD "1234";
```

5.2 Grant

Esta sentencia permitirá asignarle permisos a algún usuario que haya sido creado en el sistema. Los permisos únicamente podrán ser otorgados sobre una base de datos.

Sintaxis

```
GRANT <NOMBRE_USUARIO> ON <NOMBRE_BD>;
```

Ejemplo

```
GRANT Pedro ON Prueba1;
```

5.3 Revoke

Esta sentencia permitirá retirar los permisos sobre la base de datos a algún usuario.

Sintaxis

```
REVOKE <NOMBRE_USUARIO> ON <NOMBRE_BD>;
```

Ejemplo

```
REVOKE Pedro ON Prueba1;
```

6 Lenguaje de Manipulación de datos (DML).

Son las sentencias que permiten manipular la información almacenada en la base de datos. Las sentencias que soporta CQL son las siguientes:

6.1 Insertar

Esta sentencia se utiliza para agregar nuevos registros a una tabla. Se puede realizar de 2 maneras:

- **Inserción Normal:** Permitirá ingresar un registro a una tabla, pero debe de incluir todos los campos correspondientes a ese registro.

Sintaxis

```
INSERT INTO <NOMBRE_TABLA> VALUES '(' <LISTA_VALORES> '');
```

Ejemplo

```
INSERT INTO Estudiante VALUES ( 1, "Juan Valdez" , "Colombia" );
```

- **Inserción especial:** Permitirá ingresar un registro a una tabla, pero únicamente con los datos que le indiquemos

Sintaxis

```
INSERT INTO <NOMBRE_TABLA> '(' <LISTA_CAMPOS> ')  
VALUES '(' <LISTA_VALORES> '');
```

Ejemplo

```
INSERT INTO Estudiante( id, Nombre) VALUES ( 2, "Lilly Collins" );
```

6.2 Actualizar

Esta sentencia se utiliza para actualizar los campos de un registro ya existente:

Sintaxis

```
UPDATE <NOMBRE_TABLA> SET
```

```
Asignacion ( , Asignacion)*
```

```
[WHERE Expresion]? ;
```

Ejemplo

```
UPDATE Estudiante
```

```
SET Nombre= 'Pao',
```

```
    Edad=21
```

```
WHERE Nombre=="Paola" && Edad<18;
```

```
UPDATE UserActions
```

```
SET total = total + 2
```

```
WHERE userID = "B70DE1D0-9908-4AE3-BE34-5573E5B09F14"
```

```
&& action = "click";
```

Consideraciones

- Si la cláusula WHERE no se incluye, se actualizarán todos los registros de la tabla.

6.3 Eliminar

Esta sentencia se utiliza para eliminar los campos de un registro ya existente:

Sintaxis

```
DELETE FROM <NOMBRE_TABLA> [WHERE Expresion]?
```

Ejemplo

```
DELETE FROM Estudiante
WHERE Nombre=="Julio" && Edad<18;

DELETE FROM Estudiante;
```

Consideraciones

- Si la cláusula WHERE no se incluye, se eliminarán todos los registros de la tabla

6.4 Seleccionar

Esta sentencia permite realizar consultas sobre los datos almacenados en la base de datos. Presenta varias opciones tales como ordenar por un campo específico, mostrar únicamente un cierto número de datos, etc.

Sintaxis

```
SELECT ( Lista_Campos | * )
FROM Nombre_Tabla
[WHERE Expresion]?
[ORDER BY Nombre_Campo (ASC| DESC)?,(',' Nombre_Campo (ASC| DESC)?)*
]?
[LIMIT Expresion]?
```

Ejemplo

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT alumno.carnet FROM Estudiante

SELECT time, value
```

```
FROM events
WHERE event_type == "myEvent"
  && time > '2011-02-03'
  && time <= '2012-01-01'
```

Consideraciones

- Si no se especifica ASC O DESC en la cláusula ORDER BY, se tomará por defecto ASC.
- Si se utiliza el asterisco (*) en lugar de la lista de campos, la consulta devolverá todos los campos de la tabla en el orden que fueron creados.
- Si uno de los campos fuera un objeto, o un MAP, se deberán de mostrar una representación lineal de sus atributos, por ejemplo '{“Nombre” : “Juan” , “Edad”: 18}', esta representación queda a criterio del estudiante.
- Si uno de los campos fuera una LIST, o un SET, se deberán de mostrar una representación lineal de sus elementos, por ejemplo '[1,2,3,4]', esta representación queda a criterio del estudiante.
- Al ser una base de datos NO RELACIONAL, únicamente se podrán realizar consultas sobre una sola tabla.
- Por cada SELECT que se encuentre en el archivo de entrada, que no sea parte de una función de agregación, se deberá de crear una nueva pestaña mostrando los resultados en la aplicación cliente.

6.5 Batch

Esta sentencia se utiliza para agrupar un conjunto de sentencias DML

Sintaxis

```
BEGIN BATCH
(Insert_Statement | Update_Statement | Delete_Statement)+
APPLY BATCH;
```

Ejemplo

```
BEGIN BATCH
```



```
UPDATE users SET password == "ps22dhds" WHERE userid == "user3";
INSERT INTO users (userid, password) VALUES ("user4", "ch@ngem3c");
DELETE name FROM users WHERE userid == "user1" || userid == "user2";

APPLY BATCH;
```

Consideraciones

- Dentro de un BATCH únicamente podrán aparecer sentencias del tipo INSERT, DELETE o UPDATE para cualquier otra sentencia se debería de reportar un error.
- Si Cualquiera de las sentencias dentro del Batch tuviera un error, se debería de descartar la ejecución del Batch por completo.

6.6 Funciones de Agregación

El lenguaje CQL contara con un conjunto de funciones nativas que denominaremos funciones de agregación, estas funciones retornarán un único valor y podrán ser utilizadas como expresiones, estas son:

- **COUNT**: devuelve el número total de filas seleccionadas por la consulta.
- **MIN**: devuelve el valor mínimo del campo que especifiquemos.
- **MAX**: devuelve el valor máximo del campo que especifiquemos.
- **SUM**: suma los valores del campo que especifiquemos.
- **AVG**: devuelve el valor promedio del campo que especifiquemos.

Sintaxis

```
(COUNT| MIN| MAX| SUM| AVG) '(' '<' Select_Statement '>' )'
```

Ejemplo

```
COUNT( < SELECT Id FROM Estudiante >);
If( AVG(< SELECT EDAD FROM Estudiante >) > 18)...
```

Consideraciones

- Las funciones de MAX y MIN podrán ser utilizadas tanto en fechas, como en columnas numéricas, todas las demás funciones de agregación, únicamente se podrán utilizar sobre columnas de tipo numérico.
- Las funciones de agregación NO podrán ser utilizadas como un campo en la sentencia SELECT.
- La sentencia SELECT dentro de las funciones de agregación debe de ir encerrada entre signos de mayor y menor "<>"

6.7 Clausula Where

La cláusula where permite a las sentencias DML filtrar el número de resultados a operar. Aquí son válidas las operaciones relacionales, lógicas y aritméticas definidas en FCL.

Adicionalmente también es válido el operador **IN** el cual recibe una lista de valores separados por coma, una Lista o un Set. Esta sentencia debe ser manejada como una operación múltiple de tipo OR. Por ejemplo

```
...
WHERE carnet IN (201901, 201902, 201903)

// Es equivalente a

WHERE carnet == 201901 || carnet == 201902 || carnet == 201903

//Ejemplo con Collections
SELECT Nombre FROM Jeans
WHERE 32 IN tallas

SELECT Nombre FROM Jeans
WHERE tallas.contains(32)
```

6.8 Uso de colecciones y objetos como componentes de una tabla

Como se especificó en el lenguaje DDL, los tipos de las columnas también podrán ser objetos, por lo tanto, es permitido que se utilicen collections. Se presentan los siguientes ejemplos de la sintaxis que se utilizara para acceder a sus atributos.

6.8.1 Maps

Se crea la siguiente tabla de ejemplo para mostrar los accesos que se pueden realizar sobre maps.

```
CREATE TABLE usuarios (  
    Id string PRIMARY KEY,  
    Nombre string,  
    Favoritos Map<String, String>  
);
```

6.8.1.1 Insertar

Se utilizará la siguiente sentencia para insertar elementos a un map dentro de una tabla.

```
INSERT INTO usuarios (id, name, Favoritos)  
    VALUES ("jsmith", "John Smith", { "fruta" : "Manzana", "verdure":"Tomate"});
```

6.8.1.2 Actualizar

Para actualizar los elementos de un map se pueden utilizar las siguientes sentencias

```
// Actualiza todos los valores del mapa  
UPDATE usuarios SET Favoritos = { "fruta", "Durazno" } where id == "jsmith";  
  
// Actualiza un elemento especifico del mapa  
UPDATE usuarios SET Favoritos["fruta"] = "Durazno" where id == "jsmith";
```

6.8.1.3 Agregar

Se pueden agregar elementos a través de la siguiente sentencia

```
UPDATE usuarios SET Favoritos = Favoritos + { "película":"El Hobbit", "fruta":  
"Fresa" } WHERE id = "jsmith";
```

6.8.1.4 Eliminar

Se puede eliminar un registro de un map con las siguientes sentencias:

```
DELETE Favoritos["fruta"] FROM usuarios WHERE id == "jsmith";  
UPDATE usuarios SET Favoritos = Favoritos - {"pelicula"} WHERE id == "jsmith";
```

6.8.2 Sets

Se crea la siguiente tabla de ejemplo para mostrar los accesos que se pueden realizar sobre sets. Los valores de los sets no se pueden modificar.

```
CREATE TABLE tweets (  
    Id int,  
    texto string,  
    tags set<string>    // un set de strings  
);
```

6.8.2.1 Insertar

Se utilizará la siguiente sentencia para insertar elementos a un set dentro de una tabla.

```
INSERT INTO tweets (id, texto, tags)  
    VALUES (1, "Iniciando proyecto", { "#ConTodo", "#HoySiSale" });
```

6.8.2.2 Agregar

Se pueden agregar elementos a un set a través de la siguiente sentencia

```
// Actualiza todos los valores del set  
UPDATE tweets SET tags = { "#SaleEnVacaxD" } WHERE id = 1;  
  
// Para agregar uno o varios elementos al set  
UPDATE tweets SET tags = tags + { "#Ayuuuuda" } WHERE id = 1;
```

6.8.2.3 Eliminar

Se puede eliminar un registro de un set con las siguientes sentencias:

```
// Elimina un elemento  
UPDATE images SET tags = tags - { "#Ayuuuuda" } WHERE id = 1;
```

6.8.3 Listas

Se crea la siguiente tabla de ejemplo para mostrar los accesos que se pueden realizar sobre listas.

```
CREATE TABLE jeans (  
    id int,  
    color string,  
    marca string,  
    tallas list<string>  
);
```

6.8.3.1 Insertar

Se utilizará la siguiente sentencia para insertar elementos a una lista dentro de una tabla.

```
INSERT INTO jeans(id, color, marca, tallas)  
    VALUES( 1, "azul", "Polo", [ "S", "M", "L"] );
```

6.8.3.2 Actualizar

Para actualizar los elementos de una lista se pueden utilizar las siguientes sentencias

```
//reemplaza la lista completa  
UPDATED jeans SET tallas = [26, 28, 30] WHERE id = 1;  
  
//Para modificar el valor de un elemento en una posición específica  
UPDATE jeans SET tallas.[0] = 27 WHERE id = 1;
```

6.8.3.3 Agregar

Se pueden agregar elementos a través de la siguiente sentencia

```
UPDATE jeans SET tallas = tallas + [32, 34] WHERE id = 1;
```

6.8.3.4 Eliminar

Se puede eliminar un registro de una lista con las siguientes sentencias:

```
DELETE tallas[0] FROM jeans where id = 1;
```

6.8.4 User Types

Se crea la siguiente tabla de ejemplo para mostrar los accesos que se pueden realizar sobre los tipos definidos por el usuario.

```
CREATE TABLE Estadio (  
    Id int,  
    Joga Jugador,  
    Asi Asiento  
);
```

6.8.4.1 Insertar

Se utilizará la siguiente sentencia para insertar elementos a una lista dentro de una tabla.

```
INSERT INTO Estadio (id,juga,Asi)  
    VALUES(1, {"nombre" : "Juan", "Edad": 20}, {"fila":5, "No." : 3});
```

6.8.4.2 Actualizar

Para actualizar los elementos de una lista se pueden utilizar las siguientes sentencias

```
UPDATED Estadio SET juga.nombre = "Pedro" where id = 1;
```

7 Lenguaje de Control de Flujo (FCL)

FCL como cualquier lenguaje de programación, tiene su propia estructura, reglas de sintaxis y paradigma de programación, FCL es un lenguaje derivado de C que utiliza el paradigma de programación imperativo.

7.1.1 Paradigma de programación

El paradigma usado en FCL es el paradigma imperativo y estructurado, es decir que recurre únicamente a subrutinas y tres tipos de estructura básicas:

- **Secuencia:** Todas aquellas instrucciones que no alteran el flujo de la ejecución y sigue ejecutando en orden secuencial.
- **Instrucciones de selección:** Que alteran directamente el flujo de la ejecución dependiendo de una condición de selección.
- **Cíclicas o bucles:** son todos aquellos bucles de instrucciones con una condición inicial.

También cuenta con procedimientos, funciones y estructuras propias del lenguaje, las cuales se describirán más adelante.

7.2 Sintaxis

A continuación, se detalla la sintaxis que manejarán las sentencias de FCL.

7.2.1 Bloque de sentencias

Sera un conjunto de sentencias delimitado por llaves "{ }", cuando se haga referencia a esto querrá decir que se está definiendo un entorno local con todo y sus posibles instrucciones, estas instrucciones pueden ser del lenguaje DDL, DML o FCL, además, las variables declaradas en dicho entorno, únicamente podrán ser utilizadas en dicho entorno o bien en entornos hijos.

7.2.2 Signos de agrupación

Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizarán los signos de agrupación. Para los signos de agrupación se utilizarán los paréntesis.

Ejemplo de sintaxis

```
55 * ((432- 212) / 123)
```

7.2.3 Variables

Las variables serán unidades básicas de almacenamiento en FCL. Una variable se definirá por la combinación del símbolo “@”, un identificador, un tipo y un inicializador opcional. Además, la variable tiene un entorno que define su usabilidad.

7.2.4 Declaración de variables

Una variable deberá ser declarada antes de poder ser utilizada. Para la declaración de variables será necesario empezar con la palabra reservada definiendo el tipo de la variable, seguida del identificador que esta tendrá. La variable podrá o no tener un valor de inicialización. Es posible declarar dos o más variables a la vez, en tal caso los identificadores estarán separados por comas.

Sintaxis

```
Tipo_Dato Lista_ID [= Expresion]? ;
```

Ejemplo

```
Int @var1, @var2 = 0;

Estudiante @est = new Estudiante;

Map @var = new Map<String, Int>;
```

7.2.5 Asignación de variables

Una asignación de variable consiste en asignar un valor a una variable. La asignación será llevada a cabo con el signo igual (=).

Sintaxis

```
Assignment ::= id( "." Id) "=" <expression> ";"
```

Ejemplo

```
@var2 = 0;

@est = new Estudiante;
@est.nombre = @est2.nombre;

@var = new Map<String, Int>;

@obj.attr1.attr2.attr3 = "valor";
```


7.2.6 Valores predeterminados para las variables

Cada variable en un programa debe tener un valor antes de que se use su valor:

- Cada variable se inicializa con un valor predeterminado cuando se crea:
 - Para el tipo `int`, el valor predeterminado es cero, es decir, `0`.
 - Para el tipo `double`, el valor por defecto es cero positivo, es decir, `0.0`.
 - Para el tipo `boolean`, el valor predeterminado es `false`.
 - Para todos los tipos de referencia como `String`, `Date`, `Time`, Objetos y colecciones, el valor predeterminado es `null`.
- Cada parámetro del método o función se inicializa al valor de argumento correspondiente proporcionado por el invocador del método o función

7.2.7 Operadores aritméticos

Una operación aritmética es un conjunto de reglas que permiten obtener otras cantidades o expresiones a partir de datos específicos. Cuando el resultado de una operación aritmética sobrepase el rango establecido para los tipos de datos deberá mostrarse un error en tiempo de ejecución. Para la precedencia de operadores consultar Apéndice A.

A continuación, se definen las operaciones aritméticas soportadas por el lenguaje.

7.2.7.1 Suma

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más (+). En la Tabla 3 se encuentran las diferentes posibilidades de una suma.

Tabla 3: sistema de tipos para la suma.
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
<code>int + double</code> <code>double + int</code> <code>double + double</code>	<code>double</code>	<code>5 + 4.5 = 9.5</code> <code>7.8 + 3 = 10.8</code> <code>3.56 + 2.3 = 5.86</code>
<code>int + int</code>	<code>int</code>	<code>4 + 5 = 9</code>
<code>String + int</code> <code>int + String</code> <code>String + double</code> <code>double + String</code> <code>String + boolean</code> <code>boolean + String</code> <code>String + String</code>	<code>String</code>	<code>"1" + 2 = "12"</code> <code>1 + "2" = "12"</code> <code>"1" + 2.0 = "12.0"</code> <code>1.0 + "2" = "1.02"</code> <code>"es " + true = "es true"</code> <code>false + " es" = "false es"</code> <code>"es " + "true" = "es true"</code>

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El operador de suma (+) está sobrecargado, ya que cuando el tipo de alguno de los dos operandos sea de tipo String, se realizará una concatenación.

7.2.7.2 Resta

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos (-). En la Tabla 6 se encuentran las diferentes posibilidades de una resta.

Tabla 4: sistema de tipos para la resta
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int - double double - int double - double	double	$5 - 4.5 = 0.5$ $7.8 - 3 = 4.8$ $3.56 - 2.36 = 1.2$
int - int	int	$4 - 5 = -1$

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

7.2.7.3 Multiplicación

Operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco (*). En la Tabla 5 se encuentran las diferentes posibilidades de una multiplicación.

Tabla 5: sistema de tipos para la multiplicación
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int * double double * int double * double	double	$1 * 4.5 = 4.5$ $5.1 * 3 = 15.3$ $3.0 * 2.5 = 7.5$

<code>int * int</code>	<code>int</code>	<code>4 * 5 = -1</code>
------------------------	------------------	-------------------------

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

7.2.7.4 Potencia

Operación aritmética que consiste en una multiplicación de multiplicandos iguales abreviada. Matemáticamente tiene la siguiente forma x^n , se multiplica n (exponente) veces x (base), el operador de la potencia es el “**”, teniendo de operador izquierdo la base y de operador derecho el exponente.

En la Tabla 6 se encuentran las diferentes posibilidades de una multiplicación.

Tabla 6: sistema de tipos para la multiplicación
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
<code>Int**double</code> <code>Double**int</code> <code>Double**double</code> <code>Int**int</code>	<code>double</code>	<code>1**4.5 = 1.0</code> <code>5.1**3 = 132.650999</code> <code>3.0**2.5 = 15.588457</code> <code>4**5 = 1024.0</code>

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

7.2.7.5 Modulo

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado será el residuo de realizar la división entre el dividendo y el divisor. El operador del módulo es el signo de porcentaje (%). En la Tabla 7 se encuentran las diferentes posibilidades un módulo.

Tabla 7: sistema de tipos para el modulo
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
-----------	-----------------	----------

int % double double % int double % double	double	10 % 2.5 = 0.0 5.0 % 2 = 1.0 10.0 % 7 = 3.0
int % int	int (se toma solo la parte entera)	10 % 4 = 2

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El divisor debe ser diferente de cero, de lo contrario tendrá que reportarse un error semántico.

7.2.7.6 División

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal (/). En la Tabla 8 se encuentran las diferentes posibilidades de una división.

Tabla 8: sistema de tipos para la división
Fuente: Elaboración propia.

Operandos	Tipo resultante	Ejemplos
int / double double / int double / double	double	10 / 2.5 = 4.0 5.0 / 2 = 2.5 10.0 / 2.5 = 4.0
int / int	int (se toma solo la parte entera)	10 / 4 = 2

Consideraciones:

- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.
- El divisor debe ser diferente de cero, de lo contrario tendrá que reportarse un error semántico.

7.2.7.7 Incremento

Operador unario que incrementa el valor del operando en uno. El operador del aumento son dos signos más (++).

Sintaxis

```
@<identificador> "++"
```

*Sintaxis 1: operador postfijo de incremento
Fuente: Elaboración propia.*

```
int @contador = 10;  
int @varIncremento = @contador++; // Valor 11
```

*Ejemplo 1: operador postfijo de incremento
Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:
 1. Modificar el valor de la variable almacenada en la tabla de símbolos, aumentado en uno dicho valor.
 2. Retornar el antiguo valor de la variable, es decir, sin realizar el aumento.
- El valor retornado, podrá o no ser utilizado.

7.2.7.8 Decremento

Operador unario que disminuye el valor del operando en uno. El operador del decremento son dos signos menos (--).

Sintaxis

```
@<identificador> "--"
```

*Sintaxis 2: operador postfijo de decremento
Fuente: Elaboración propia.*

```
int @contador = 10;  
int @varIncremento = @contador--; // Valor 9
```

*Ejemplo 2: operador postfijo de decremento
Fuente: Elaboración propia.*

Consideraciones:

- El operando deberá ser un identificador de una variable previamente definida.
- Este operador realizara dos acciones específicas:

1. Modificar el valor de la variable almacenada en la tabla de símbolos, disminuyendo en uno dicho valor.
 2. Retornar el antiguo valor de la variable, es decir, sin realizar el decremento.
- El valor retornado, podrá o no ser utilizado.

7.2.7.9 Asignación y Operación

Esta funcionalidad realizará la asignación y operación hacia la variable con la que se está operando. Estos operadores únicamente podrán aplicarse a valores numéricos en lenguaje FCL.

Operador	Ejemplo	Descripción
+=	@variable += 10;	Realizará la suma de 10 con la variable y el resultado será asignado a la variable.
*=	@Variable *= 20;	Realizará la multiplicación de 20 con la variable y el resultado será asignado a la variable.
-=	@Variable -= 15;	Realizará la resta de 15 a la variable y el resultado será asignado a la variable.
/=	@Variable /= 30;	Realizará la división entre 30 a la variable y el resultado será asignado a la variable.

7.2.8 Operadores relacionales

Una operación relacional es una operación de comparación entre dos valores, siempre devuelve un valor de tipo lógico (`boolean`) según el resultado de la comparación. Una operación relacional es binaria, es decir siempre tiene dos operandos.

7.2.8.1 Operadores de comparación numérica y de igualdad

En la tabla 9 se muestran los ejemplos de los operadores relacionales. Para la precedencia de operadores consultar Apéndice A.

Tabla 9: operadores relacionales
Fuente: Elaboración propia.

Operador relacional	Operador relacional	Ejemplos de uso
<pre>int [>,<,>=,<=] double double [>,<,>=,<=] int char [>,<,>=,<=] double double [>,<,>=,<=] double int [>,<,>=,<=] int Date [>,<,>=,<=] Date Time [>,<,>=,<=] Time</pre>	>, <, >=, <=	<pre>5 > 4.5 = true 4.5 < 3 = false 45.2 > 52.2 = false 4 <= 4 = true '1997-11-11' <= '1997-11-12' = true</pre>
<pre>int [==,!=] double double [==,!=] int double [==,!=] double int [==,!=] int String [==,!=] String boolean [==,!=] Boolean Date [==,!=] Date Time [==,!=] Time</pre>	==, !=	<pre>5 == 4.5 = false 4.5 != 3 = true 45.2 == 52.2 = false 4 != 4 = false "abc" != "ABC" = true '10:50:50' != '10:50:59' = true</pre>

Consideraciones:

- Será válido comparar datos numéricos (entero, decimal) entre sí, la comparación se realizará utilizando el valor numérico con signo de cada dato.
- Es válido comparar igualdad y diferencia (==, !=) para los Tipos definidos por el usuario, fechas, horas y Strings,
- Cualquier otra combinación será inválida y deberá lanzar un error de semántica.

7.2.9 Operaciones lógicas

Las operaciones lógicas son expresiones matemáticas cuyo resultado será valor lógico (boolean). Las operaciones lógicas se basan en el álgebra de Boole. Para la precedencia de operadores consultar Apéndice A

En la tabla 10 se muestra las tablas de verdad para las operaciones lógicas. Los operadores disponibles son:

- Suma lógica o disyunción, conocida como OR
- Multiplicación lógica o conjunción, conocida como AND (&&)
- Suma exclusiva, conocido como XOR (^)
- Negación, conocida como NOT (!)

Tabla 10: tabla de verdad para operadores booleanos
Fuente: Elaboración propia.

OPERANDO A	OPERANDO B	A B	A && B	A ^ B	!A
FALSE	false	false	false	false	true
FALSE	true	true	false	true	true
TRUE	false	true	false	true	false
TRUE	true	true	true	false	false

Consideraciones:

- Ambos operadores deberán ser de tipo booleano.

7.2.10 Operador ternario

Este operador retornará una expresión en función de una condición lógica. Para la precedencia de operadores consultar Apéndice A

Sintaxis

`<ternary> ::= <expression1>"?"<expression2>":"<expression3>`

Sintaxis 3: operador ternario
Fuente: Elaboración propia.

Si la condición es verdadera entonces el valor resultante será expression1, de lo contrario si la condición fuera falsa el valor retornado será expression2.

Consideraciones:

- El valor implícito de `expression1` deberá ser de tipo booleano.
- Si `expression1` es verdadera se retornará el valor implícito de `expression2`, de lo contrario se retornará el valor de la `expression3`.

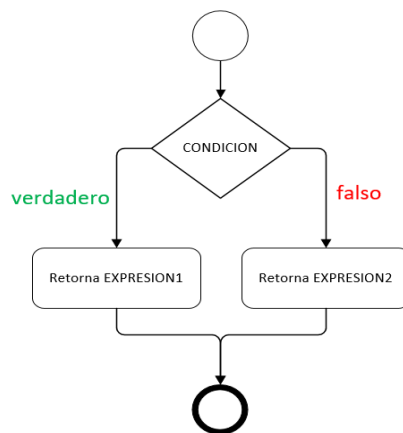


Figura 2: diagrama de flujo para un operador ternario
Fuente: Elaboración propia.

7.2.11 Sentencias de Selección

Las sentencias de selección en FCL, son todas aquellas que alteran el flujo secuencial de la ejecución, siempre dependerán de una condición y las sentencias de selección de FCL son las siguientes:

7.2.11.1 Sentencia If

Esta sentencia de selección bifurcará el flujo de ejecución ya que proporciona control sobre dos alternativas basadas en el valor lógico de una expresión (condición).

Sintaxis

```
if (CONDICION) BLOQUE_SENTENCIAS
```

Ejemplo

```
if (@variable > 0) {  
    @var1 = 0;  
}
```

7.2.11.2 Sentencia else

Esta sentencia de selección capturará una sentencia de If anteriormente falsa o reevaluará una nueva sentencia de selección.

Sintaxis

```
(SENTENCIA_If)  
else  
( BLOQUE_SENTENCIAS | SENTENCIA_IF )
```

Ejemplo

```
if (@variable > 0) {  
    @var1 = 0;  
} else if (@variable < 0) {  
    @var1 = 1;  
} else {  
    @var1 = 2;  
}
```

7.2.11.3 Sentencia switch

Esta sentencia de selección será la bifurcación múltiple, es decir, proporcionará control sobre múltiples alternativas basadas en el valor de una expresión de control.

La sentencia selecciona compara con el valor seguido de cada caso, y si coincide, se ejecuta el bloque de sentencias asociadas a dicho caso, hasta encontrar una instrucción detener.

Sintaxis

```
switch(EXPRESION) {  
  case EXPRESION: BLOQUE_SENTENCIAS [detener;]  
  (case EXPRESION: BLOQUE_SENTENCIAS [detener;])*  
  [default: BLOQUE_SENTENCIAS [detener;]]  
}
```

Ejemplo

```
switch (@y /50) {  
  case 1:  
    // sentencias que se ejecutarán si @y/50 = 1.  
  case 2:  
    // sentencias que se ejecutarán si @y/50 = 2.  
  default:  
    // sentencias que se ejecutarán si @y/50 no coincide ningún // caso.  
}
```

Consideraciones

- Si i es el caso actual y n el número de casos, si la expresión del caso i llegará a coincidir con la expresión de control (para $i < n$), se ejecutarán todos los bloques de sentencias entre los casos i y n mientras no se encuentre la instrucción detener.
- Tanto la expresión de control como las expresiones asociadas a cada uno de los casos deberá ser de tipo primitivo, es decir, carácter, entero, decimal o booleano.
- Si ninguno de los casos coincide con la expresión de control se ejecutará el bloque de sentencias asociadas a defecto

7.2.12 Sentencias cíclicas o bucles

Este tipo de sentencias en FCL son todas aquellas que incluyen un bucle restringido sobre una condición, las sentencias cíclicas que soporta FCL son las siguientes:

7.2.12.1 Sentencia While

Esta sentencia cíclica ejecutará todo el bloque de sentencias asociado mientras la condición que la restringe sea verdadera, de lo contrario el programa seguirá su flujo secuencial.

Sintaxis

```
While(EXPRESION) (BLOQUE_SENTENCIAS)
```

Ejemplo

```
while (@cuenta < 5) {  
    @cuenta++;  
}
```

7.2.12.2 Sentencia Do While

Esta sentencia cíclica ejecutará todo el bloque de sentencias una vez y se seguirá ejecutando mientras la condición sea verdadera (*1 a n veces*), de lo contrario el programa continuará con su flujo de ejecución normal.

Sintaxis

```
Do (BLOQUE_SENTENCIAS)  
While(EXPRESION) ;
```

Ejemplo

```
Do{  
    @cuenta++;  
}while (@cuenta < 5);
```

7.2.12.3 Sentencia For

Esta sentencia cíclica para permitirá inicializar o establecer una variable como variable de control, el ciclo tendrá una condición que se verificará en cada iteración, luego se deberá definir una operación que actualice la variable de control cada vez que se ejecuta un ciclo para luego verificar si la condición se cumple. Las partes de esta sentencia serán las siguientes:

- **Inicialización:** Declaración o asignación.
- **Condición:** Expresión booleana que se evaluará cada iteración para decidir si se ejecuta el bloque de sentencias o no.
- **Actualización:** Sentencia que actualiza la variable de control ya sea con una asignación o una expresión de incremento o decremento

Sintaxis

```
For( <Inicializacion> ; <Condicion> ; <Actualizacion>) (BLOQUE_SENTENCIAS)
```

Ejemplo

```
for (int @cuenta = 0; @cuenta < 5; @cuenta++) {  
    // Bloque de sentencias a ejecutarse  
}
```

7.2.13 Collections

FCL tendrá la posibilidad de manejar conjuntos de estructuras dinámicas y genéricas que facilitarán el manejo de elementos de diferentes tipos, las collection que soportará FCL serán los siguientes:

7.2.13.1 Map

Un collection map es un set ordenado de pares clave-valor, donde las claves son únicas y sirven para mantener el mapa ordenado. Esta collection, su clave debe ser de tipo primitivo, debe ser único y el valor contenido puede ser de cualquier tipo, e incluso otra collection.

Para instanciar un collection tipo Map existen dos formas,

- Una instancia vacía, únicamente declarando los tipos de la clave y valor
- Una instancia con valores, declarando entre corchetes una lista de elementos clave-valor, es importante resaltar que las claves y valores de estos elementos deben ser del mismo tipo entre sí.

Sintaxis para instancia vacía

```
Map @var = new Map<CQL_TIPO_PRIMITIVO, CQL_TIPO >;
```

Sintaxis para una instancia con elementos

```
Map @var = [<clave : valor> (<clave : valor>)* ]
```

Ejemplo

```
Map @var1 = new Map<int, string>;  
Map @var2 = [ 1 : "Julius" , 2 : "Luis" , 3 : "Rainman"];
```

El collection map contará con un conjunto de funciones propias para el manejo de sus datos, las funciones son las siguientes:

- **Insert(clave,valor):** función especializada para la inserción de datos en la lista, contará con dos parámetros donde el primero estará almacenada la clave y en el segundo será el valor de dicho elemento.
- **Get(clave):** función que retorna el valor de la lista que contenga tal clave, en caso no exista un valor con dicha clave retornará null.
- **Set(clave,valor):** función especializada para la modificación de datos en la lista, contará con dos parámetros donde el primero estará almacenada la clave y en el segundo será el valor con que lo actualizará, en caso no exista dicha clave se lanza un error .
- **Remove(clave):** función que eliminará el elemento que contenga dicha clave en la lista.
- **Size():** retornará la cantidad de elementos que contiene la collection.
- **Clear():** limpiará el collection dejándolo con cero elementos.
- **Contains(elemento):** Se le pasara como parámetro la clave buscada y retornará true si hay un valor de la lista que contenga tal clave, caso contrario retornará false.

7.2.13.2 List

Una lista es una colección de elementos homogéneos, no necesariamente únicos, donde los elementos son ordenados según como se van insertando en la lista. La posición de los elementos inicia en 0.

Para instanciar un collection tipo List existen dos formas,

- Una instancia vacía, únicamente declarando los tipos del valor
- Una instancia con valores, declarando entre corchetes una lista de elementos, es importante resaltar que todos estos elementos deben ser del mismo tipo.

Sintaxis para instancia vacia

```
List @var = new List<CQL_TIPO >;
```

Sintaxis para una instancia con elementos

```
List @var = [<valor> (<valor>)* ]
```

Ejemplo

```
List @var1 = new List<string>;  
List @var2 = [ "Julius" , "Luis" , "Rainman" ];
```

El collection list contará con un conjunto de funciones propias para el manejo de sus datos, las funciones son las siguientes:

- **Insert(valor):** función especializada para la inserción de datos en la lista, contará con un único parámetro que será el valor para insertar. El valor se agrega al final.
- **Get(posición):** función que retorna el valor de la lista que este en la posición que recibe de parámetro.
- **Set(posición,valor):** función especializada para la modificación de datos en la lista, contará con dos parámetros donde el primero estará almacenada la posición a modificar y en el segundo será el valor con que lo actualizará, en caso no exista dicha clave se lanza un error .
- **Remove(posición):** función que eliminará el elemento que contenga la lista en la posición que recibe de parámetro.
- **Size():** retornará la cantidad de elementos que contiene el collection.
- **Clear():** limpiará el collection dejándolo con cero elementos.
- **Contains(elemento):** Retornara true si el elemento está en la lista, caso contrario retornará false. En el caso de los objetos se comparará la instancia del objeto.

7.2.13.3 Set

Un set de datos es una colección de elementos homogéneos, estos datos son únicos y no se pueden repetir, además, son ordenados según su valor de menor a mayor, en caso de que el valor sea un collection o un user-type no se ordenarán, insertando solo al final, pero si se verificará que no estén repetidos.

Para instanciar un collection tipo Set existen dos formas,

- Una instancia vacía, únicamente declarando los tipos del valor
- Una instancia con valores, declarando entre corchetes una lista de elementos, es importante resaltar que todos estos elementos deben ser del mismo tipo.

Sintaxis para instancia vacía

```
Set @var = new Set<CQL_TIPO >;
```

Sintaxis para una instancia con elementos

```
Set @var = [<valor> (,<valor>)* ]
```

Ejemplo

```
Set @var1 = new Set<string>;  
Set @var2 = [ " Julius" , "Luis" , "Rainman" ];
```

El collection Set contará con un conjunto de funciones propias para el manejo de sus datos, las funciones son las siguientes:

- **Insert(valor):** función especializada para la inserción de datos en la lista set, contará con un único parámetro que será el valor para insertar.
- **Get(posicion):** función que retorna el valor de la lista set que este en la posición que recibe de parámetro.
- **Set(posición,valor):** función especializada para la modificación de datos en la lista, contará con dos parámetros donde el primero estará almacenada la posición a modificar y en el segundo será el valor con que lo actualizará, en caso no exista dicha clave se lanza un error .
- **Remove(posicion):** función que eliminará el elemento que contenga la lista en la posición que recibe de parámetro.
- **Size():** retornará la cantidad de elementos que contiene el collection.
- **Clear():** limpiará el collection dejándolo con cero elementos.
- **Contains(elemento):** Retornara true si el elemento está en la lista, caso contrario retornará false. En el caso de los objetos se comparará la instancia del objeto.

7.2.14 Funciones

Las funciones en FCL contienen un conjunto de instrucciones que se puede invocar, pasando un número fijo de valores como parámetros y retornan un valor en específico.

Sintaxis

```
<Tipo_Dato> <Nombre_Funcion> ((<TipoParametro><NombreParametro>)* )  
(BLOQUE_SENTENCIAS)
```

Ejemplo

```
Int factorial(int @n){  
    if (@n == 0) {  
        return 1;  
    } else {  
        return @n * factorial(n - 1);  
    }  
}
```

Consideraciones

- El tipo de dato de la función puede ser un tipo primitivo, collection, cursor o un user type.
- Se pueden definir de cero a muchos parámetros, siempre separados por coma.
- Los tipos de los parámetros pueden ser de tipo primitivo, collection o un user type.
- Las funciones solo pueden retornar un único valor y siempre tienen que retornarlo.
- Las funciones en CQL **no** se guardan en la base de datos, únicamente existen en el ámbito global de donde acaban de ser creadas, si se cierra y abre CQL, estas funciones ya no existirán.

7.2.14.1 Sobrecarga de funciones

FCL tendrá la propiedad de soportar la sobrecarga de funciones, esto quiere decir que podrá tener diferentes funciones con el mismo identificador, pero varía según las propiedades de sus parámetros, FCL permitirá la sobrecarga si cumplen con las siguientes características:

- **Diferente cantidad de parámetros:** se considerarán diferentes las funciones si cuentan con diferente cantidad de parámetros independientemente si tienen el mismo identificador como nombre.
- **Tipo de parámetros.** se considerarán diferentes las funciones si cuentan con diferente tipo de parámetro en al menos un parámetro, esto quiere decir que, si tiene el mismo identificador y la misma cantidad de parámetros, pero alguno de los parámetros varia su tipo con el de la otra función, se permite la sobrecarga.

7.2.14.2 Llamadas a funciones

Las llamadas a funciones se pueden realizar en cualquier punto de la ejecución ya sea como una sentencia por aparte o como parte de una expresión, para esta última es necesario verificar el tipo de la función, al momento de llamar a una función esta ejecuta toda la función en un nuevo entorno y al finalizarlo sigue su ejecución normal desde el punto donde fue llamada.

Una llamada está conformada por una lista de valores opcionales que serán los valores que tomarían los parámetros de la función. Se debe de corroborar que la llamada a función que se desea ejecutar exista y tenga la misma cantidad de parámetros y además todos del tipo correcto.

Sintaxis

```
<NOMBRE_FUNCION> (<LISTA_VALORES>*)
```

Ejemplo

```
Funcion_ejemplo(1,2,3,4,5,"aaaa");
```

Consideraciones

- Debido a la sobrecarga de funciones se debe verificar la cantidad y tipos de parámetros a pasar con las funciones existentes.

7.2.15 Procedimientos

Los procedimientos en FCL contienen un conjunto de instrucciones que se puede invocar, pasando un número fijo de valores como parámetros, pero se diferencian de las funciones debido a que los procedimientos pueden retornar más de un valor y estos si están guardados en las bases de datos.

Sintaxis

```
Procedure <Nombre_Procedure>  
((<TipoParam><NombreParam>)* ) , ((<TipoRetorno><NombreRetorno>)* )  
(BLOQUE_SENTENCIAS)
```

Ejemplo

```
Procedure Ejemplo_Procedure(int @n, int @p), (int @retorno1, int @ret2){  
    if (@n == 0) {  
        return 1, 2;  
    } else {  
        return @n, @n *2;  
    }  
}
```

Consideraciones

- Cada tipo de retorno del procedure puede ser un tipo primitivo, collection, cursor o un user type.
- Se pueden definir de cero a muchos parámetros y de cero a muchos retornos, siempre separados por coma.
- Los tipos de los parámetros pueden ser de tipo primitivo, collection o un user type.
- En caso de que no se tenga parámetros de retorno, la función puede acabar sola o bien con un return sin valor.
- Los procedimientos en CQL **si** se guardan en la base de datos.

7.2.15.1 Sobrecarga de Procedimientos

FCL tendrá la propiedad de soportar la sobrecarga de procedimientos, esto quiere decir que podrá tener diferentes procedimientos con el mismo identificador, pero varía según las propiedades de sus parámetros, FCL permitirá la sobrecarga si cumplen con las siguientes características:

- **Diferente cantidad de parámetros:** se considerarán diferentes los procedimientos que cuentan con diferente cantidad de parámetros independientemente si tienen el mismo identificador como nombre.
- **Tipo de parámetros.** se considerarán diferentes los procedimientos si cuentan con diferente tipo de parámetro en al menos un parámetro, esto quiere decir que, si tiene el mismo identificador y la misma cantidad de parámetros, pero alguno de los parámetros varia su tipo con el de la otra función, se permite la sobrecarga.

7.2.15.2 Llamadas a procedimientos

Las llamadas a procedimientos se pueden realizar en cualquier punto de la ejecución, pero únicamente como una sentencia por aparte o como una captura de retornos.

Una llamada está conformada por una lista de valores opcionales que serán los valores que tomarían los parámetros de la función. Se debe de corroborar que la llamada a función que se desea ejecutar exista y tenga la misma cantidad de parámetros y además todos del tipo correcto.

Sintaxis sentencia por aparte

Para este tipo de llamada no se captura los posibles retornos que contiene el procedimiento, únicamente se realiza la llamada y ejecución del procedimiento

```
Call <NOMBRE_PROC> (<LISTA_VALORES>*)
```

Ejemplo

```
Call Proc_ejemplo(1,2,3,4,5,"aaaa");
```

Sintaxis captura de retornos

Para este tipo de llamada se debe tener control de los retornos que puede devolver el procedimiento, para ello se debe realizar una lista de variables, previamente declaradas, igualadas a la llamada del procedimiento, las variables de las listas toman el valor de los retornos exactamente en el mismo orden que se declararon.

```
<LISTA_VARIABLES> = Call <NOMBRE_PROC> (<LISTA_VALORES>*)
```

Ejemplo

```
Procedure Ejemplo_Procedure(int @n, int @p), (int @retorno1, int @ret2){  
    if (@n == 0) {  
        return 1, 2;  
    } else {  
        return @n, @n * 2;  
    }  
}
```

```
Int @A;  
Int @B;  
  
@A, @B = Call Ejemplo_Procedure (1,2);
```

Consideraciones

- Debido a la sobrecarga de procedimientos se debe verificar la cantidad y tipos de parámetros a pasar con las funciones existentes.
- Para la captura de retornos se debe verificar que cada variable de captura debe ser del mismo tipo que el parámetro de retorno que le corresponda.

7.2.16 Funciones nativas sobre cadenas

CQL ofrece distintas funciones nativas, entre estas están las que se pueden aplicar a una variable de tipo cadena, que analizan la cadena contenida en la variable y devuelven un valor en específico dependiendo de la función, estas funciones nativas son:

7.2.16.1 *Length*

Esta función analiza la cadena y devuelve un entero que corresponde a la longitud total de la cadena.

Sintaxis

```
Variable.lenght();
```

Ejemplo

```
String @var = "Hola Mundo";  
Int tam = @var.Length(); // tam = 10
```

7.2.16.2 *toUpperCase*

Esta función analiza la cadena y devuelve una nueva cadena, convirtiendo cada carácter a su equivalente en mayúscula.

Sintaxis

```
Variable.toUpperCase();
```

Ejemplo

```
String @var = "Hola Mundo";  
String @var2 = @Var.toUpperCase(); // var2 = "HOLA MUNDO"
```

7.2.16.3 toLowerCase

Esta función analiza la cadena y devuelve una nueva cadena, convirtiendo cada carácter a su equivalente en minúscula.

Sintaxis

```
Variable.toLowerCase();
```

Ejemplo

```
String @var = "Hola Mundo";  
String @var2 = @Var.toLowerCase(); // var2 = "hola mundo"
```

7.2.16.4 startsWith

Esta función recibe de parámetro una cadena y analiza si la cadena a la cual se le aplica la función comienza con la cadena del parámetro, devolviendo un valor booleano, siendo verdadero si comienza y falso sino comienza.

Sintaxis

```
Variable. startsWith (<Cadena>);
```

Ejemplo

```
String @var = "Hola Mundo";  
boolean @var2 = @Var. startsWith ("Hola");// var2 = true
```

7.2.16.5 endsWith

Esta función recibe de parámetro una cadena y analiza si la cadena a la cual se le aplica la función termina con la cadena del parámetro, devolviendo un valor booleano, siendo verdadero si termina y falso sino termina.

Sintaxis

```
Variable. endsWith (<Cadena>);
```

Ejemplo

```
String @var = "Hola Mundo";  
boolean @var2 = @Var. endsWith ("Hola");// var2 = false
```

7.2.16.6 *subString*

Esta función recibe de parámetro dos enteros, simbolizando una posición de inicio y una cantidad de caracteres respectivamente, analiza la cadena a la cual se le aplica la función devolviendo una nueva cadena con los caracteres desde la posición de inicio hasta la posición de inicio más la cantidad de caracteres. Las cadenas inician con posición 0.

Sintaxis

```
Variable. subString (<posInicio>, <Cantidad>);
```

Ejemplo

```
String @var = "Hola Mundo";  
String @var2 = @Var. subString (0,4); //var2 = "Hola"
```

7.2.17 Funciones nativas de abstracción

Entre las diferentes funciones nativas también se encuentran las de abstracción de información, estas se aplican sobre los tipos de dato Time y Date, debido a que son tipos de datos que se puede extraer información, como el año, mes, día, hora, minutos y segundos, o bien también son funciones que traen información del servidor. Las funciones nativas son:

7.2.17.1 *GetYear*

Esta función nativa puede ser aplicada únicamente en variables de tipo Date y devuelve un entero con el valor del año de dicha variable.

Sintaxis

```
Variable. getYear();
```

7.2.17.2 *GetMonth*

Esta función nativa puede ser aplicada únicamente en variables de tipo Date y devuelve un entero con el valor del mes de dicha variable.

Sintaxis

```
Variable. getMonth();
```

7.2.17.3 *GetDay*

Esta función nativa puede ser aplicada únicamente en variables de tipo Date y devuelve un entero con el valor del día de dicha variable.

Sintaxis

```
Variable. getDay();
```

Ejemplos

```
Date fechaActual = '1999-12-10';  
  
Int year = fechaActual.getYear(); // 1999  
Int mes = fechaActual.getMonth(); // 12  
Int dia = fechaActual.getDay(); // 10
```

7.2.17.4 *GetHour*

Esta función nativa puede ser aplicada únicamente en variables de tipo Time y devuelve un entero con el valor del año de dicha variable.

Sintaxis

```
Variable. getHour();
```

7.2.17.5 *GetMinuts*

Esta función nativa puede ser aplicada únicamente en variables de tipo Time y devuelve un entero con el valor del mes de dicha variable.

Sintaxis

```
Variable. getMinuts();
```

7.2.17.6 *GetSeconds*

Esta función nativa puede ser aplicada únicamente en variables de tipo Time y devuelve un entero con el valor del día de dicha variable.

Sintaxis

```
Variable. getSeconds();
```

Ejemplos

```
Time horaActual = '10:45:59';  
  
Int hora = horaActual.getHour(); // 10  
Int minutos = horaActual.getMinuts(); // 45  
Int segundos = horaActual.getSeconds(); // 59
```

7.2.17.7 *Today*

Función nativa que trae la fecha actual del servidor, devuelve un valor de tipo Date.

Sintaxis

```
Today();
```

Ejemplos

```
Date fechaActual = today();
```

7.2.17.8 *Now*

Función nativa que trae la hora actual del servidor, devuelve un valor de tipo Time.

Sintaxis

```
Now();
```

Ejemplos

```
Time horaActual = Now();
```

7.2.18 Sentencias de transferencia

FCL soportará tres sentencias de salto o transferencia: break, continue y return. Estas sentencias transferirán el control a otras partes del programa y se podrán utilizar en entornos especializados.

7.2.18.1 Sentencia Break

La sentencia break tendrá dos usos:

- Terminar la ejecución de una sentencia switch
- Terminar la ejecución de una sentencia cíclica.

Sintaxis

```
Break;
```

Ejemplo

```
while (@cuenta < 5) {  
    @cuenta++;  
    if (@cuenta == 3) {  
        break;  
    }  
}
```

Consideraciones:

- Deberá verificarse que la sentencia break aparezca únicamente dentro de una sentencia cíclica o dentro de una sentencia switch.
- En el caso de sentencias cíclicas, la sentencia break únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia cíclica que la contiene.
- En el caso de sentencia switch, la sentencia break únicamente detendrá la ejecución del bloque de sentencias asociado a la sentencia case que la contiene.

7.2.18.2 Sentencia continue

La sentencia continue se utilizará para transferir el control al principio del ciclo, es decir, continuar con la siguiente iteración.

Sintaxis

```
Continue;
```

Ejemplo

```
while (@cuenta < 5) {  
    @cuenta++;  
    if (@cuenta == 3) {  
        continue;  
    }  
}
```

Consideraciones:

- Deberá verificarse que la sentencia continue aparezca únicamente dentro de un ciclo y afecte solamente las iteraciones de dicho ciclo.

7.2.18.3 Sentencia return

La sentencia return se empleará para salir de la ejecución de sentencias de una función y devolver un valor o valores. Tras la salida del método se vuelve a la secuencia de ejecución del programa al lugar de llamada de dicho método.

Sintaxis

```
Return;
```

Ejemplo

```
Int factorial(int @n){  
    if (@n == 0) {  
        return 1;  
    } else {  
        return @n * factorial(n - 1);  
    }  
}
```

Consideraciones:

- Debe verificarse que en funciones la sentencia return, retorne una expresión del mismo tipo del que fue declarado en dicha función o procedure.
- Un return sin valor podría venir si el procedure no tiene parámetros de retorno y este únicamente detendrá la ejecución del procedure.
- Debe verificarse que la sentencia return retorne un único valor en funciones y todos los valores necesarios en un procedure.
- Debe verificarse que la sentencia return este contenida únicamente en funciones o procedimientos.

7.2.19 Cursores

Un cursor es una estructura de control que se utiliza para el recorrido y procesamiento de los registros del resultado de una consulta. Se define de la siguiente manera:

Sintaxis

```
CURSOR @ID IS Select_Statement;
```

Ejemplo

```
CURSOR @c1 IS Select * From Estudiante;
```

El Cursor se encargará de almacenar en memoria una copia del resultado de una consulta para reutilizarla cada vez que lo necesitemos, y accederemos a él por medio de la instrucción FOR EACH

Sintaxis

```
FOR EACH (Lista_Campos) IN nombre_cursor BLOQUE_SENTENCIAS
```

La lista de campos corresponde a las columnas resultantes de la sentencia select

Ejemplo

```
FOR EACH (int @Carnet, String @Nombre) IN @c1  
    Log(Nombre);
```

Para poder utilizar el cursor, es necesario que ejecutemos la sentencia OPEN, de esta manera el cursor ejecutará la consulta y obtendrá los datos.

Sintaxis

```
OPEN @ID;
```

Ejemplo

```
OPEN @c1;
```

Para liberar la memoria utilizada por el cursor, se utilizará la instrucción CLOSE, a partir de este momento el cursor ya no se podrá utilizar y de lo contrario se reportará un error en tiempo de ejecución.

Sintaxis

```
CLOSE nombre;
```

Ejemplo

```
CLOSE @c1;
```

Consideraciones:

- Se debe de validar que el número de parámetros que recibe el cursor sea el mismo número de columnas que retorna la consulta.
- Si se ejecuta la sentencia OPEN sobre un cursor abierto, los datos deberán volver a cargarse.
- Si se ejecuta la sentencia OPEN sobre un cursor cerrado, el cursor deberá de ejecutar nuevamente la consulta para obtener los resultados.
- Un cursor permanecerá cerrado hasta que se ejecute la sentencia OPEN.

7.2.20 Función LOG

Log es una función primitiva, parte de las funcionalidades intrínsecas de CQL. A través de esta función podemos visualizar mensajes en la consola de CQL-Client.

Sintaxis

```
LOG (" expresión ");
```

Ejemplo

```
LOG ( "Hola" + " " + "Mundo");  
  
Date @fecha = '2019-01-01'  
LOG("Hoy es " + @fecha);  
  
Int @contador;  
  
LOG(@contador);
```

7.2.21 Sentencia throw

Una sentencia throw hace que se lance una excepción. El resultado es una transferencia inmediata de control que puede aparecer en cualquier bloque de sentencias mientras que se encuentre dentro del cuerpo de una sentencia try que capte el valor arrojado. Si no existe tal sentencia, entonces se detiene la ejecución del programa.

Sintaxis

```
Throw new <TIPO_EXCEPCION>;
```

Ejemplo

```
throw new ArithmeticException;
```

7.2.22 Sentencia try Catch

Una sentencia try ejecuta un bloque. Si se lanza una excepción y la sentencia try tiene una cláusula catch que pueden atraparlo, entonces el control se transferirá a la cláusula catch.

Sintaxis

```
Try (BLOQUE_SENTENCIAS)  
Catch(<TIPO_EXCEPCION>) (BLOQUE_SENTENCIAS)
```

Ejemplo

```
int divisor = 0;  
int dividendo = 10;  
try {  
    int cociente == dividendo / divisor;  
} catch (ArithmeticException ae){  
    Log (e.message) ;  
}
```

8 Lenguaje Unificado de Paquetes (LUP)

El lenguaje LUP es un lenguaje de paquetes que será el encargado de manejar la comunicación entre el cliente y el servidor de base de datos. Al ser un lenguaje de comunicación interna entre el cliente y el servidor, es necesario que el estudiante implemente una nueva pestaña tipo consola en la que se pueda ver todos los paquetes enviados y recibidos para validar la correcta aplicación de estos.

Para el funcionamiento de la aplicación, los paquetes se pueden dividir en dos grupos, los que se envían del cliente al servidor y las respuestas del servidor.

8.1 Solicitudes del cliente al servidor:

Son todos los paquetes que se enviarán desde el cliente, hacia el servidor con la información ingresada por el usuario, los datos de inicio y fin de sesión, etc.

8.1.1 Paquete de Inicio de Sesión

Cada vez que un usuario haga un intento de iniciar sesión, se deben de validar las credenciales a través de un paquete de inicio de sesión. Si las credenciales coinciden, el servidor retornará SUCCESS y el usuario podrá hacer uso de la aplicación, en caso contrario, retornará FAIL y deberá ingresar nuevamente sus credenciales.

Envío del Cliente al Servidor	Respuesta del Servidor
[+LOGIN] [+USER] Luis [-USER] [+PASS] 1234 [-PASS] [-LOGIN]	[+LOGIN] [SUCCESS] [FAIL] [-LOGIN]

8.1.2 Paquete de Fin de Sesión

El cliente deberá de notificar al servidor el fin de la sesión.

Envío del Cliente al Servidor	Respuesta del Servidor
[+LOGOUT] [+USER] Luis [-USER] [-LOGOUT]	[+LOGOUT] [SUCCESS] [FAIL] [-LOGOUT]

--	--

8.1.3 Paquete de Consulta:

Es el encargado de enviar toda la información desde el editor hacia el servidor de base de datos. Además, envía el usuario que realizó la consulta, para poder validar que tenga permiso a acceder a esa información.

Envío del Cliente al Servidor	Respuesta del Servidor
<pre>[+QUERY] [+USER] Luis [-USER] [+DATA] SELECT * FROM Employee; UPDATE Employee SET apellido = "RODRIGUEZ" WHERE nombre = —'ANTONIO'; [-DATA] [-QUERY]</pre>	<pre>[+DATA] <table> <tr> <th>Nombre</th><th>Apellido</th> </tr> <tr> <td>Pedro</td><td>Perez</td> </tr> </table> [-DATA] [+MESSAGE] Registro modificado exitosamente. [-MESSAGE]</pre>

8.2 Respuestas del servidor al cliente:

Son todos los paquetes de respuesta sobre una petición que el cliente haya iniciado. El servidor únicamente se comunicará con el cliente para enviarle una respuesta, no envía solicitudes.

8.2.1 Paquete de Datos

Es el paquete se enviará como respuesta a un paquete de consulta enviado desde el cliente. Este paquete contendrá toda la información de una consulta de tipo SELECT. Es decisión del alumno el formato en que desea retornar la información, pero se recomienda una tabla de html para facilitar la creación de la pestaña en el lado del cliente.

Respuesta del Servidor
<pre>[+DATA] <table> <tr> <th>Nombre</th><th>Apellido</th> </tr> <tr> <td>Pedro</td><td>Perez</td> </tr> </table></pre>

[-DATA]

8.2.2 Paquete de Error

Es el paquete que se enviará como respuesta a una petición de consulta del cliente, este contendrá toda la información de un error encontrado durante la fase de análisis.

Respuesta del Servidor
[+ERROR]
[+LINE]
15
[-LINE]
[+COLUMN]
44
[-COLUMN]
[+TYPE]
(Semantico sintactico lexico)
[-TYPE]
[+DESC]
No se reconoce el caracter '\$;
[-DESC]
[-ERROR]

8.2.3 Paquete de Mensaje

Imprimirá en consola todos los mensajes de la sentencia Log encontrados durante el análisis de un archivo CQL enviado del cliente al servidor, así como todas las notificaciones que el estudiante considere necesarias

Respuesta del Servidor
[+MESSAGE]
El resultado de la operación es: 20
[-MESSAGE]
[+MESSAGE]
El usuario prueba21 se ha creado correctamente.
[-MESSAGE]

8.2.4 Paquete de Estructura

Es Cliente deberá de solicitar al servidor un paquete de estructura. Este le permitirá crear un árbol que le muestre al usuario todas las bases de datos, tablas, objetos, procedimientos, etc. Que fueron creados. Se mostrarán únicamente los objetos CQL, en los que el usuario tiene permisos.

Envío del Cliente al Servidor	Respuesta del Servidor
[+STRUC] [+USER] Luis [-USER] [-STRUC]	[+DATABASES] [+DATABASE] [+NAME] Prueba1 [-NAME] [+TABLES] [+TABLE] [+NAME] Prueba1 [-NAME] [+COLUMNS] Column1 [-COLUMNS] [-TABLE] [+TABLE] MAESTRO [-TABLE] [-TABLES] [+TYPES] [+TYPE] [+NAME] Persona [-NAME] [+ATTRIBUTES] nombre [-ATTRIBUETS] [-TYPE] [+TYPE] DOMICILIO [-TYPE] [-TYPES] [+PROCEDURES] CreateStudent [-PROCEDURES] [-DATABASE] [-DATABASES]

El estudiante debe agregar todos los tags que considere necesaria para desplegar la información de todos los elementos, tales como nombre, columnas, tipos, etc.

9 El Lenguaje de Almacenamiento de Datos (CHISON)

El lenguaje chison, derivado de chinese JSON, es un lenguaje de notación que objetos que le permitirá al servidor de bases de datos almacenar su información y garantizar la persistencia de esta a lo largo del tiempo. TODOS los datos contenidos en las distintas bases de datos se deben de almacenar al ejecutarse una sentencia commit, y al ejecutarse la sentencia rollback retornara al último estado de consistencia de la información, en este caso sería el último commit.

9.1 Notación

Al ser un sistema de bases de datos NO-SQL completamente orientado a objetos, las bases de datos, tablas, usuarios, permisos, etc. Se podrían tomar como objetos también, por lo tanto, se define la siguiente sintaxis para el almacenamiento de los datos:

9.1.1 Datos Primitivos

Para cada uno de los tipos de datos primitivo en formato chison, se tiene una notación específica:

9.1.1.1 Números

Los números tanto en formato entero como decimal se almacenarán sin comillas.

9.1.1.2 Cadenas

Se almacenarán entre comillas dobles.

9.1.1.3 Booleanos

Se almacenarán con las palabras reservadas True y False sin utilizar comillas dobles.

9.1.1.4 Date

Se almacenarán utilizando comillas simples en formato 'yyyy-mm-dd'.

9.1.1.5 Time

Se almacenarán utilizando comillas simples en formato 'hh-mm-ss'.

9.1.1.6 Objetos

Los objetos tendrán una definición especial dentro del lenguaje chison, se utilizará lo siguiente:

Sintaxis

```
< "Nombre_atributo1" = valor , "Nombre_atributo2" = valor >
```

Ejemplo

```
"Nombre_Objeto" = < "Nombre" = "Pao" , "Edad" = 21, "Fecha_Nac" = '1997-12-21'>
```

9.1.1.7 Listas

Las listas utilizan una definición diferente a la de los objetos ya que cada uno de sus valores no tiene un nombre propio. Se utilizará la siguiente sintaxis:

Sintaxis

```
[ atributo1, atributo2, atributo3... , atributoN ]
```

Ejemplo

```
"Nombre_Lista" = [1, 2, 3, 4, 5]
```

9.2 Definición de los archivos

En el lenguaje chison, los archivos contarán con una notación específica que le permita recuperar los datos de una forma fácil y efectiva.

9.2.1 Archivo Principal

Existirá un archivo principal desde donde se iniciará la recuperación de la información, este archivo almacenará todos los bases de datos creadas, así como los usuarios y sus permisos. Su sintaxis se define de la siguiente forma:

Sintaxis

```
$<  
  "DATABASES"= [ Lista_BasesDatos ],  
  "USERS"= [ Lista_Usuarios ]  
>$
```

Ejemplo

```
$<  
  "DATABASES"= [  
    <  
      "NAME"= "Database 1",  
      "DATA"= [ Lista_CQL-Type]  
    >,  
  ]  
>$
```

```

<
  "NAME"= "Database 2",
  "DATA"= [ Lista_CQL-Type ]
>,
<
  "NAME"= "Database 3",
  "DATA"= [ Lista_CQL-Type ]
>
],
"USERS"= [
  <
    "NAME"="Luis",
    "PASSWORD"= "1234",
    "PERMISSIONS"= [
      <
        "NAME": "Database 1"
      >,
      <
        "NAME": "Database 2"
      >
    ]
  >,
  <
    "NAME" = "Fernando",
    "PASSWORD" = "5678",
    "PERMISSIONS" = []
  >
]>$

```

Consideraciones

- Nótese que las bases de datos aparecen primero que los usuarios, de otra manera no se podría determinar si las bases de datos existen o no al momento de crear los usuarios.
- Si algún usuario tuviera permiso de acceder a una base de datos que no existe, se reportara el error al momento de cargar la información en la base de datos.

9.2.1.1 Usuarios

Por cada usuario en el sistema, se deberá de crear un objeto de tipo usuario en el lenguaje chison, este objeto además de contar con el nombre de usuario y la contraseña contará con una lista de los permisos que tiene el usuario.

Sintaxis

```
<
  "NAME" = "nombre",
  "PASSWORD" = "pass",
  "PERMISSIONS" = [ Lista_Permisos]
>
```

Ejemplo

```
<
  "NAME"="Luis",
  "PASSWORD"= "1234",
  "PERMISSIONS"= [
    <
      "NAME" = "Database 1"
    >,
    <
      "NAME" = "Database 2"
    >
  ]
>
```

9.2.1.2 Bases de datos

Por cada base de datos, se deberá de crear un objeto de tipo base de datos en el lenguaje chison, este objeto además de contar con el nombre de la base de datos contara con una lista de los objetos que la base de datos tiene almacenada. Estos se detallarán más adelante.

Sintaxis

```
<
  "NAME" = "nombre",
  "Data" = [ Lista_Permisos]
>
```

Ejemplo

```
<
  "NAME"="Database 1",
  "DATA"= []
>
```

9.2.1.2.1 Tablas

Una tabla pertenece únicamente a una base de datos, por lo tanto, su definición sería como un objeto dentro de la lista "DATA" de la base de datos. Su sintaxis es la siguiente:

Sintaxis

```
<
  "CQL-TYPE" = "TABLE",
  "NAME" = "nombre",
  "COLUMNS" = [ Lista_Columnas],
  "DATA" = [ Lista_Datos ]
>
```

Ejemplo

```
<
  "NAME" = "Estudiante",
  "COLUMNS" = [
    <
      "NAME" = "ID",
      "TYPE" = "Counter",
      "PK" = TRUE
    >,
    <
      "NAME" = "Nombre",
      "TYPE" = "String",
      "PK" = FALSE
    >
  ],
  "DATA" = [
    <
      "ID" = 1,
      "Nombre" = "Diana"
    >,
    <
      "ID" = 2,
      "Nombre" = "Eliel"
    >
  ]
>
```

Columnas

Una tabla contiene una lista de columnas, las cuales también definiremos como un objeto dentro del lenguaje:

Sintaxis

```
<
  "NAME" = "nombre",
  "TYPE" = "Tipo_Dato",
  "PK" = (TRUE | FALSE)
>
```

Ejemplo

```
<
  "NAME" = "ID",
  "TYPE" = "Counter",
  "PK" = TRUE
>
```

9.2.1.2.2 User Types

Los user types son los objetos definidos por el usuario dentro del lenguaje CQL, estos se almacenarán como un identificador con una lista de atributos:

Sintaxis

```
<
  "CQL-TYPE" = "OBJECT",
  "NAME" = "nombre",
  "ATTRS" = [ Lista_Atributos ]
>

//Atributo

<
  "NAME" = "nombre",
  "TYPE" = "Tipo_Dato"
>
```

Ejemplo

```
<
  "CQL-TYPE" = "OBJECT",
  "NAME" = "Alumno",
  "ATTRS" = [
    <
      "NAME" = "Nombre",
      "TYPE" = "String"
    >,
    <
      "NAME" = "Edad",
      "TYPE" = "Int"
    >
  ]
>
```

9.2.1.2.3 Procedimientos

Los procedimientos son una de las utilidades más grandes del lenguaje chison, por lo tanto, es indispensable que dichos objetos sean persistentes en el tiempo a través del tiempo. Su sintaxis se definió como:

Sintaxis

```
<
  "CQL-TYPE" = "PROCEDURE",
  "NAME" = "Procedure1"
  "PARAMETERS" = [Lista_Parametros],
  "INSTR" = "lista de sentencias del procedimiento"
>

// Parametro

<
  "NAME" = "nombre",
  "TYPE" = "Tipo_Dato",
  "AS" = (IN|OUT)
>
```


Ejemplo

```
<
  "CQL-TYPE" = "PROCEDURE",
  "NAME" = "Procedure1"
  "PARAMETERS" = [
    <
      "NAME" = "Nombre",
      "TYPE" = "String",
      "AS" = IN
    >,
    <
      "NAME" = "ID_In",
      "TYPE" = "Int",
      "AS" = IN
    >,
    <
      "NAME" = "Edad",
      "TYPE" = "String",
      "AS" = OUT
    >,
  ],
  "INSTR" = " Select edad from Estudiante where id=ID_In "
>
```

Consideraciones

- Para todos los tipos de objetos que como uno de sus atributos se solicite el campo TYPE, los tipos de datos permitidos serán los mismos que el lenguaje FCL, es decir INT, DOUBLE, STRING, DATE, TIME, BOOLEAN, COUNTER y cualquier otro tipo definido en CQL, tal como un nombre propio o una colección.
- Si el tipo de dato fuera una colección, el tipo se definiría como List<Tipo_Dato>, Map<Tipo_Dato>, Set<Tipo_Dato>, en donde Tipo_Dato puede ser cualquiera de los especificados en el primer punto de estas consideraciones.
- Si el tipo de dato fuera un objeto, se definiría con el nombre propio del objeto y debe de verificarse que dicho objeto si exista, de lo contrario se reportaría un error en tiempo de ejecución.

9.3 Importar Archivo

En el lenguaje chison, cualquier definición de un objeto (<Attrs>), o lista de objetos(<Attrs>,<Attrs>), podrá ser reemplazado por un literal de importación, esta importación hará referencia a un archivo con formato chison y ruta relativa a la carpeta que se esté analizando. Se deberá de insertar dicho archivo en el lugar donde se encuentre la importación. Su sintaxis es la siguiente:

Sintaxis

```
{ nombre.chison }
```

Ejemplo

```
$<
"DATABASES"= [
    ${ databases.chison }$
],
"USERS"= [
    ${ users.chison }$
]
>$
```

10 Excepciones

Cuando un programa viola las restricciones semánticas del lenguaje de programación, el compilador señala este error como una excepción. CQL tendrá un conjunto de excepciones predefinidas para mejorar la captura y detalle de estas. Las excepciones que manejará CQL son las siguientes:

- **TypeAlreadyExists:** Excepción que se disparará al momento de querer crear un [User Type](#) con un nombre ya existente, esta excepción no se dispara si la creación tiene la directiva **IF NOT EXISTS**.
- **TypeDontExists:** Excepción que se disparará al momento de querer [alterar](#) o [eliminar](#) un tipo de dato inexistente.
- **BDAAlreadyExists:** Excepción que se disparará al momento de querer crear una [base de datos](#) con un nombre ya existente, esta excepción no se dispara si la creación tiene la directiva **IF NOT EXISTS**.
- **BDDontExists:** Excepción que se disparará al momento de querer [usar](#) o [eliminar](#) una base de datos inexistente.
- **UseBDException:** Excepción que se disparará al momento de querer ejecutar una sentencia en la cual sea necesario tener en uso una base de datos y ninguna este en [uso](#).

- **TableAlreadyExists:** Excepción que se disparará al momento de querer crear una [Tabla](#) con un nombre ya existente, esta excepción no se dispara si la creación tiene la directiva **IF NOT EXISTS**.
- **TableDontExists:** Excepción que se disparará al momento de querer [alterar](#), [truncar](#), [eliminar](#), [insertar](#), [actualizar](#) en una tabla inexistente.
- **CounterTypeException:** Excepción que se disparará al momento de querer insertar o actualizar un dato en una columna de tipo [Counter](#).
- **UserAlreadyExists:** Excepción que se disparará al momento de querer crear un [usuario](#) con un nombre ya existente.
- **UserDontExists:** Excepción que se disparará al momento de querer [asignar](#) o [quitar](#) permisos a un usuario inexistente.
- **ValuesException:** Excepción que se disparará al momento de querer [insertar](#) datos en una tabla y los valores no coincidan con los tipos o bien no hay suficientes valores.
- **ColumnException:** Excepción que se disparará al momento de querer utilizar una columna inexistente en una operación.
- **BatchException:** Excepción que se disparará al momento de no poder ejecutar todo el [batch](#) completo.
- **IndexOutOfRangeException:** Excepción que se disparará al momento de querer acceder a una posición inválida dentro de un [collection](#).
- **ArithmeticException:** Excepción que se disparará cuando se evalúa una expresión que viola las reglas definidas sobre las operaciones [aritméticas](#).
- **NullPointerException:** Excepción que se disparará cuando se desea utilizar miembros de un objeto que apunta a null.
- **NumberReturnsException:** Excepción que se disparará cuando se trata de un [procedures](#) y la cantidad de [variables de retorno](#) no coincide con la cantidad de retornos del procedure.
- **FunctionAlreadyExists:** Excepción que se disparará al momento de querer crear una [función](#) ya existente.
- **ProcedureAlreadyExists:** Excepción que se disparará al momento de querer crear un [procedure](#) ya existente.
- **ObjectAlreadyExists:** Excepción que se disparará al momento de querer crear una [instancia](#) con identificador ya existente.

11 Apéndice A: Precedencia y asociatividad de operadores

Para saber el orden jerárquico de las operaciones se define la siguiente precedencia de operadores. La precedencia de los operadores irá de menor a mayor según su aparición en la tabla 12.


A continuación, se presenta la precedencia de operadores lógicos.

Tabla 11: precedencia y asociatividad de operadores lógicos
Fuente: Elaboración propia.

NIVEL	OPERADOR	DESCRIPCION	ASOCIATIVIDAD
12	()	paréntesis	Izquierda
11	++	incremento postfijo	no
	--	decremento postfijo	asociativo
10	-	menos unario	Derecha
	!	not	
9	* / %	multiplicativas	Izquierda
8	+ -	aditivas	Izquierda
	+	concatenación de cadenas	
7	< <=	relacionales	no
	> >=		asociativo
6	==	igualdad	izquierda
	!=		
5	^	Xor	izquierda
4	&&	And	Izquierda
3		Or	Izquierda
2	? :	ternario	Derecha
1	=	asignación	derecha

12 Manejo de errores

La herramienta deberá de ser capaz de identificar todo tipo de errores al momento de interpretar los lenguajes de entrada, tanto CQL como CHISON. Para el lenguaje CQL, se solicita que, a través de los paquetes de error recibidos en el cliente como respuesta a un paquete de consulta, el estudiante cree un archivo con formato HTML donde reporte todos los errores encontrados.



Fila	Columna	Tipo	Descripción
10	32	Lexico	No se esperaba el caracter #
43	12	Semantico	La variable nombre no ha sido declarada

Para el lenguaje CHISON, se debe implementar una tabla tipo “log” con en el servidor de base de datos para que almacene los errores, dicha tabla tendrá el nombre de “Errors” y contara con las siguientes columnas:

1. Numero de error
2. Tipo Error. Con los valores LEXICO, SINTACTICO, SEMANTICO
3. Descripción, mostrando la excepción que se provoco
4. Fila
5. Columna
6. Fecha
7. Hora

Al ser una tabla como todas las demás, se podrán realizar consultas sobre ella para visualizar todos los errores obtenidos durante la lectura de la información. Una consulta como ‘Select * from Errors where tipo == “SEMANTICO” ’ debería de dar un resultado similar a:

Numero	Tipo	Descripción	Fila	Columna
1	Semántico	El tipo de dato Casa no fue declarado	3	1
2	Semántico	La base de datos Prueba no existe	74	15
3	Semántico	Llave primaria ID duplicada	89	12

12.1 Tipos de errores

Los tipos de errores se deberán de manejar son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos

12.2 Contenido mínimo de tabla de errores

La tabla de errores debe contener como mínimo la siguiente información:

- Línea: Número de línea donde se encuentra el error.
- Columna: Número de columna donde se encuentra el error.
- Tipo de error: Identifica el tipo de error encontrado. Este puede ser léxico, sintáctico o semántico.
- Descripción del error: Dar una explicación concisa de por qué se generó el error.

12.3 Método de recuperación

- Léxicos y sintácticos, modo pánico.
- Semánticos, a nivel de instrucción.

13 Archivos de Entrada

Podrán encontrar los archivos de entrada para mayor comprensión del proyecto en este [enlace](#).

14 Entregables y Restricciones

14.1 Entregables

1. Código fuente
2. Aplicación funcional
3. Gramáticas
4. Manual técnico
5. Manual de usuario

Deberán entregarse todos los archivos necesarios para la instalación en el servidor de aplicaciones, así como el código fuente, la gramática y la documentación. La calificación del proyecto se realizará con los archivos entregados en la fecha establecida. El usuario es completa y únicamente responsable de verificar el contenido de los entregables. La calificación se realizará sobre archivos ejecutables. Se proporcionarán archivos de entrada al momento de calificación.

14.2 Restricciones

1. Para los analizadores del lenguaje Lup en el lado del servidor se deberá utilizar Irony y C#, para el lado del cliente se deberá utilizar Jison y Javascript.
2. Para los analizadores de Chison y CQL se deberá utilizar Irony y C#.
3. Copias de proyectos tendrán de manera automática una nota de 0 puntos y serán reportados a la Escuela de Ingeniería en Ciencias y Sistemas los involucrados.
4. Todos los lenguajes implementados no hacen distinción entre mayúsculas y minúsculas, es decir no son case sensitive.

14.3 Requisitos mínimos

Los requerimientos mínimos del proyecto son funcionalidades del sistema que permitirán un ciclo de ejecución básica, para tener derecho a calificación se deben cumplir con lo siguiente:

2 Descripción

2.1.5 Login

2.1.6 Panel de información

2.1.7 Salida de datos y Consola

2.1.8 Modos de edición

2.1.8.1 Principiante

2.1.8.2 Avanzado

2.4 Definiciones generales de CQL

2.4.1 Identificadores

- 2.4.2 Case Insensitive
 - 2.4.3 Comentarios
 - 2.4.4 Valor nulo
- 2.5 Tipos de dato
 - 2.5.1 Tipos primitivos
 - 2.5.2 Secuencias de escape
 - 2.5.3 User Types
 - 2.5.4 Casteos primitivos en CQL
- 3 Lenguaje de Definición de Datos (DDL).
 - 3.1 Crear Base de datos
 - 3.2 Sentencia Use
 - 3.4 Crear Tabla
 - 3.4.1 Tipo Dato Counter
 - 3.4.2 Llaves Primarias
 - 3.4.3 Tipos de datos para definición de columnas
 - 3.5 Alter Table
- 4 Lenguaje de Control de Transacciones (TCL).
 - 4.1 Commit
 - 4.2 Rollback
- 5 Lenguaje de Control de Datos (DCL).
 - 5.1 Crear Usuario
- 6 Lenguaje de Manipulación de datos (DML).
 - 6.1 Insertar
 - 6.2 Actualizar
 - 6.4 Seleccionar
 - 6.6 Funciones de Agregación
 - 6.7 Clausula Where
 - 6.7.1 Operaciones Lógicas
 - 6.8 Uso de colecciones y objetos como componentes de una tabla
 - 6.8.3 Listas
 - 6.8.4 User Types

7 Lenguaje de Control de Flujo (FCL)

7.2 Sintaxis

7.2.3 Variables

7.2.4 Declaración de variables

7.2.5 Asignación de variables

7.2.6 Valores predeterminados para las variables

7.2.7 Operadores aritméticos

7.2.8 Operadores relacionales

7.2.9 Operaciones lógicas

7.2.12 Sentencias de Selección

7.2.12.1 Sentencia If

7.2.12.2 Sentencia else

7.2.13 Sentencias cíclicas o bucles

7.2.13.1 Sentencia While

7.2.13.2 Sentencia For

7.2.14 Collections

7.12.14.1 Lists

7.2.15 Funciones

7.2.16 Procedimientos

7.2.19 Sentencias de transferencia

7.2.19.1 Break

7.2.19.2 Return

7.2.20 Cursores

7.2.21 Log

8 Lenguaje Unificado de Paquetes (LUP)

8.1 Solicitudes del cliente al servidor:

8.1.1 Paquete de Inicio de Sesión

8.1.2 Paquete de Fin de Sesión

8.1.3 Paquete de Consulta:

8.2 Respuestas del servidor al cliente:

8.2.1 Paquete de Datos

- 8.2.2 Paquete de Error
- 8.2.3 Paquete de Mensaje
- 8.2.4 Paquete de Estructura
- 9 El Lenguaje de Almacenamiento de Datos (CHISON)
 - 9.1 Notación
 - 9.1.1 Datos Primitivos
 - 9.2 Definición de los archivos
 - 9.2.1 Archivo Principal
 - 9.3 Importar Archivo
- 10 Excepciones
- 11 Apéndice A: Precedencia y asociatividad de operadores
- 12 Manejo de errores

14.4 Entrega del proyecto

1. La entrega será virtual y por medio de la plataforma de la clase.
2. La entrega de cada uno de los proyectos es individual.
3. Para la entrega del proyecto se deberá cumplir con todos los requerimientos mínimos.
4. No se recibirán proyectos después de la fecha ni hora de la entrega estipulada.
5. Es responsabilidad del estudiante asegurarse que los archivos enviados funcionen y puedan desplegarse correctamente para su propia calificación.
6. La entrega del proyecto será mediante un archivo comprimido de extensión rar o zip.
7. Entrega del proyecto:

Miércoles 18 de septiembre de 2019 a las 23:59 horas