Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2024

LABORATORIO 3

Laboratorio:	Evaluación de consultas.
Publicación:	Miércoles 15 de mayo.
Ayudantías:	Viernes 17 & 24 de mayo.
Entrega:	**Viernes 31 de mayo hasta las 23:59 horas**.

Descripción del laboratorio

En este laboratorio exploraremos cómo IIC3413DB [1] procesa consultas SQL. Siguiendo la estructura de un motor SQL explicada en las clases, IIC3413DB usa tres componentes para procesar las consultas (la implementación se encuentra en la carpeta `query` del código):

- `parser`, cual contiene la definición de la gramática de un fragmento de SQL, parser de consultas que cumplan con esta gramática, el pre-processor cual genera el primer plan lógico, y un query rewriter cual permite mejorar el plan inicial.
- `optimizer`, cual se encarga de elegir el orden de joins y generar un plan físico; y
- `executor`, cual define la estructura general para ejecutar la cosulta mediante un plan físico, e contiene la implementación de operadores físicos particulares (e.g. algoritmos para el join, la selección, etc.).

A continuación explicaremos algunos detalles de cada componente.

Parser y plan lógico

La componente `parser` del IIC3413DB se encarga de parsear consultas y generar el plan lógico. La implementación se encuentra en la carpeta `/IIC3413DB/src/query/parser`. El archivo `parser.h` contiene el flujo de esta componente de nuestro sistema. La secuencia de operaciones es la siguiente:

1. **Gramática, lexer y parser (carpeta `grammar`)**. Primero nos preocuparemos de parsear la consulta. Para automatizar la tarea de parsing de consultas y su análisis léxico, ocuparemos la librería/herramienta ANTLR [2]. ANTLR permite definir los tokens léxicos de nuestro lenguaje (ver `IIC3413Lexer.g4` en la carpeta `/IIC3413DB/src/query/parser/grammar/`) y una gramática libre de contexto para las consultas permitidas (`IIC3413Parser.g4` en la misma carpeta)[1]. Con estos dos archivos, y ejecutando el script `generate.sh`, ANTLR genera todo lo necesario para armar árbol de parseo para nuestras consultas. (Cómo referfencia, este árbol es creado por el archivo `parser.h` en el comando `tree = parser.root();`). Adicionalmente, ANTLR nos genera la signatura de las clases visitor que se pueden ocupar para navegar sobre el árbol de parseo para armar un plan lógico inicial.

[1] Nuestra implementación cuenta con un fragmento reducido de SQL. En nuestras consultas, solo podemos tener una `AND` de condiciones dentro de un `WHERE` y no soportamos consultas anidadas o corelacionadas.

2. **Visitors y un primer plan lógico (carpeta `query_preprocessor`)** Luego de generar el árbol de parseo, `query_preprocessor` recorre este árbol usando los distintos visitors para generar el plan lógico inicial (el mismo explicado en las clases cual consiste de una proyección, seguida por una selección y un producto cruz de las tablas). Durante este proceso la variable `current_logical_plan` de la clase `QueryPreprocessor` colecciona los elementos necesarios para armar el plan (variables por proyectar, condiciones, relaciones, atributos, aliases para tablas (palabra `AS` en SQL), etc.). Los visitors son un patrón de diseño que nos permite navegar la estructura y cambiarnos de una clase a otra usando polimorfismo sin tener que re-implementar cada clase que vamos a ocupar. La estructura del plan lógico, dependiendo de si se trata de un plan de selección, proyección, etc. está descrita en la carpeta `logical_plan`. La expresiones que pueden aparecer en un `WHERE` tienen su propia estructura y sus visitors son definidos de manera especial (subcarpeta `logical_plan/expr`). Se les recomienda revisar el código para empezar a entender cómo los visitors crean un plan lógico.

3. **Reescritura de consultas (carpeta `query_rewriter`)** Ahora que ya contamos con un primer plan lógico, lo optimizaremos un poco usando las reglas de reescritura de consultas. Para contar con un código acotado, en este laboratorio las únicas dos operaciones que se harán es agrupar las relaciones que hacen el join y hacer push de selecciones hacía las hojas del árbol lógico. Estas dos operaciones se hacen en `build_join.h` y `push_selection.h` respectivamente. Efectivamente, `build_join.h` va a definir cuales tablas hacen join entre ellas, y no ocupar el producto cartesiano en este caso, mientras `push_selection.h` pone las condiciones junto con los operadores/relaciones que los usan. Al terminar con este paso, contamos con un plan lógico más detallado que el original.

Ayudantía 1: El viernes 17 de Mayo se hará una ayudantía sobre el parser y visitor pattern, el patrón de diseño que es estándar para hacer parsing de cualquier tipo de cosas. Si quieren estudiar sobre el visitor pattern/ANTLR por su propia cuenta, se recomienda el siguiente recurso:

https://tomassetti.me/getting-started-antlr-cpp/

Optimizer, Executor y plan físico

Ahora que ya contamos con un plan lógico, lo vamos a convertir a un plan físico usando la clase `Optimizer` (ver el archivo `optimizer.h` en la carpeta `src/query/optimizer`). El plan físico es simplemente un iterador sobre los resultados de nuestra consulta cual implementa la clase abstracta `QueryIter` (ver `query_iter.h` en la carpeta `src/query/executor`). El plan físico es construido por la clase `Optimizer` de nuevo usando un visitor. El iterador que define nuestro plan físico está compuesto de varios iteradores que implementan la clase abstracta `QueryIter`, uno para cada operador de álgebra relacional soportado por nuestro fragmento de SQL (para ver sus implementaciones; o sea, los operadores físicos que los definen, consulten `src/query/executor`). Estos `QueryIter`s se componen siguiendo la estructura del plan lógico permitiendo una ejecución pipelined.

Nuestro `Optimizer` toma el plan lógico generado por el parser, y empieza visitar distintos operadores para generar sus respectivos `QueryIter`s. Una tarea importante cual se realiza en este proceso es definir el orden de los joins y productos cruz. En nuestro caso, optimizer simplemente genera joins de une left-deep plan usando el orden de relaciones mencionado en la clausula `FROM` de nuestra consulta y nada más (i.e. no hay una estimación del costo). Durante este proceso, cada operador entrega un `QueryIter` cual permite iterar sobre los resultados de este sub-árbol. Una observación importante es que en el caso de selección, su visitor debe iterar sobre las expresiones de la clausula `WHERE` de nuestra consulta, y llamar los visitors para generar un evaluador de la expresión sobre la tupla considerada. Igualmente, el operador físico en `selection.h` deben considerar todas las expresiones que entran a esta selección y validarlas todas (acuerden que solo tenemos `AND` de condiciones en `WHERE`) – ver la implementación del `next()` en `selection.h`.

Valores en operadores: Para tener una evaluación eficiente de las consultas, nuestros iteradores del plan físico no van a copiar valores cuando pasan de un operador físico al otro. Efectivamente, solo las hojas de nuestro plan físico, las que representan las relaciones guardadas en la base de datos, tendrán acceso directo

a las tuplas. Todos los niveles arriba de las hojas solamente tendrán acceso a *punteros* a los valores que requieren, y no los valores mismos. Para esto en `record.h` definimos la clase `RecordRef` la cual ahora no guarda valores de una tupla (cómo un vector), sino solamente punteros a valores (de varias tuplas a la ves). Para dar un ejemplo, consideren la consulta:

```
SELECT R.a
FROM R, S
WHERE R.b = S.b
```

Árbol lógico para esta consulta tiene cuatro nodos: (i) la raíz $\pi_{R.a}$; (ii) el operador $\bowtie_{R.b=S.b}$; y (iii) dos hojas R y S. En nuestro plan físico, solo las hojas R y S tendrán el acceso a `RealtionIter` (de las tareas 1 y 2) cual nos permite conseguir las tuplas desde el heap file de nuestra tabla, mientras que al operador $\pi_{R.a}$ y $\bowtie_{R.b=S.b}$ se le pasa solo una vector de `RecordRef`s a valores de las columnas $R.a, R.b$ y $S.b$, las cuales necesitamos para calcular el join y devolver la tupla a $\pi_{R.a}$ (a este operador se les devuelve solo $R.a$ dado que no necesita los otros valores).

Ayudantía 2: El viernes 24 de Mayo se hará una ayudantía sobre el los visitors de optimizer/executor, explicando la estructura del código.

Tareas para hacer en este laboratorio

Problema 1: `LIKE` [**2 puntos**]. Este problema les pide completar la implementación del operador físico para la condición `LIKE` en el `WHERE` de nuestras consultas SQL. Quiere decir que el objetivo es poder ejecutar las consultas de tipo:

```
SELECT R.a
FROM R
WHERE R.c LIKE "%hola_"
```

Dado que se les pide implementar solo el operador físico, la parte del parser o generar el plan lógico ya viene implentado dentro del código. Lo único que deben completar es la implementación del método:

```
const Value& eval()
```

de la clase `ExprLike` ubicada en `executor/expr/expr_like.h`. En el caso de este método en particular, deben devolver un `Value&` interpretado cómo 0 cuando no hay match, y 1 en el caso que si hay match. Esto se debe a la ausencia del valor `bool` en el motor, y el hecho que algunas expresiones pueden devolver valores concretos. Por ejemplo, las expresiones cómo $R.c$ cual aparecen en un `WHERE` deben devolver el valor de la columna c de la tupla actual en la tabla R, y, por lo tanto, nuestras expresiones siempre devuelven un `Value&`. En su implementación, pueden asumir que el constructor de la clase:

```
ExprLike(std::unique_ptr<Expr> child, std::string&& _pattern)
```

fue llamado de manera correcta y que `child` contiene el puntero a la expresión cual define el valor por comparar (e.g. $R.c$ en nuestro ejemplo arriba), y `_pattern` contiene el patrón de búsqueda. Para acotar la tarea un poco, nuestro patrón de búsqueda usa las siguientes suposiciones:

- Solo consideramos caracteres ASCII.
- El carácter especial % solo aparece al inicio o al final del patrón. Recuerden que % reemplaza cualquier secuencia de caracteres (zero o más caracteres).
- El carácter especial _ aparece en cualquier lugar del patrón. Por ejemplo `R.a LIKE h_o_la` es una condición válida. Recuerden que _ reemplaza precisamente un carácter cualquiera.

Problema 2: `BETWEEN` [**4 puntos**]. En este problema deben agregar soporte para el operador `BETWEEN` de SQL a nuestro sistema. Esto quiere decir que el objetivo es soportar las consultas del tipo:

```
SELECT a
FROM R
WHERE a BETWEEN 27 AND 48"
```

aquí estamos asumiendo que la tabla `R` contiene el atributo `a` y que este sí es un `INT`. En su implementación `BETWEEN` debe tener la forma `column BETWEEN low AND high`, dónde:

- `column` es `a` (nombre de un atributo) o `R.a` (tabla con atributo).
- `low` y `high` pueden ser `INT` o `STR`, pero siempre los dos deben tener el mismo tipo.
- El tipo de dato de `column` es igual a tipo de dato `low` y `high`.

A cambio de caso de `LIKE`, su soporte para `BETWEEN` debe ser desarrollado desde el zero. Esto quiere decir que deben modificar los siguientes partes del sistema (lista no necesariamente exhaustiva):

1. Deben agregar los detalles del operador a nuestro lexer y a la gramática que soportamos (y luego deben ejecutar `generate.sh` en la carpeta `grammar`).
2. Al `query_preprocessor` deben agregar un visitor para la expresión del tipo `BETWEEN`. Aquí deben colectar el nombre de la columna y valores `low` y `high`, tal cómo validar que `low` y `high` son del mismo tipo que la columna (que la columna sí existe en la tabla ya se chequea por los métodos implementados). Recuerden también que en el `logical_plan/expr` tendrán que agregar un `ExprPlan` para `BETWEEN`.
3. En el `optimizer` agreguen el soporte para `BETWEEN`.
4. En `executor` agreguen el soporte necesario para `BETWEEN`. Una cosa qué definitivamente tendrán que agregar una implementación de la clase abstracta `Expr`.

Evaluación del Problema 2: La evaluación de este problema se hará en dos partes:

- **1. Parser y plan lógico** [**1.5 puntos**] Aquí se va a validar que su implementación puede parsear el string de la consulta correctamente y generar un plan lógico. Se revisará cómo manejan errores descritos más arriba (incompatibilidad del tipo de datos, etc.) y si el plan lógico es correcto.
- **2. Resultado de la ejecución** [**2.5 puntos**] Para este parte de la tarea se validará si su implementación de los operadores físicos es correcta; quiere decir si produce los resultados esperados.

Problema Bonus: `BETWEEN` **con ISAM** [**1 punto en cualquier lab**]. En este bonus su optimizador debe poder detectar si existe un índice ISAM para el atributo usado en `BETWEEN` y en este caso usar el iterador de ISAM (Laboratorio 2) para entregar los resultados. Es importante observar que para una solución correcta su implementación de `BETWEEN` no puede funcionar cómo una implementación de la clase abstracta `Expr` cómo en el problema 2, sino que debe aceder directamente al índice ISAM. Adicionalmente, si no existe una índice adecuado, la solución del problema 2 debe seguir funcionando. Este bonus les entrega un punto para agregar a la tarea (sí, pueden tener 8 en una tarea y se considerará para su promedio final).

Bug hunt #1 [**+0.1 puntos en la nota del lab**]. Si encuentren un bug no-trivial en nuestra implementación *y lo reportan a través de issues de gitHub* se les asignará una décima a la nota del laboratorio. Sí, pueden contar con una nota mayor que 7 en este caso (e.g. 7.2) y esta nota se considerará para el promedio de los labs.

Bug hunt #2 [**+0.1 puntos en la nota del curso**]. Si encuentren un bug no-trivial *y proponen una solución* para este bug, se les agregará una décima a la nota final del curso. Su solución debe ser enviada por correo a `vrdomagoj@uc.cl`. (En mi corazón pueden tener una nota mayor que 7 para el curso, pero para la UC la nota máxima solo puede ser un 7.)

Código

El código base está disponible en [1] en la rama `T3`. Asegúrense que están usando la última versión del sistema! Los tests se publicarán en el mismo repositorio. Cómo parte del código, también publicaremos la solución de tarea 2 (cual sirve solo para el bonus). El código también contiene un cliente que les permite ejecutar las consultas sin tener que modificar un `.cc`. Este también permite imprimir planes (lógicos y físicos). Instrucciones están en el readme del código.

Evaluación y entrega

El día límite para la entrega de esta tarea será el viernes 31 de mayo a las 23:59 horas. Para ello se utilizará el formulario de canvas. *Su entrega debe contener todo el código de la carpeta* `src` *y nada más, y debe compilar cuando se les agregan otras carpetas en el repositorio. El código de la carpeta* `src` *debe ser entregado cómo un archivo* `.zip`. Por último, la evaluación será en base a varios test que su solución debe responder.

Ayudantía y preguntas

El día viernes 24 de mayo se realizará una ayudantía donde se darán más detalles sobre IIC3413DB y el laboratorio. Para preguntas se pide usar el foro del curso o issues del GitHub.

Referencias

[1] Carlos Rojas, Diego Bustamante, Vicente Calisto, Tristan Heuer. IIC3413DB. https://github.com/DiegoEmilio01/IIC3413.

[2] Terence Parr. ANTLR. https://www.antlr.org/.