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

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2020

LABORATORIO 3

Laboratorio:	Evaluación de operadores.
Publicación:	Viernes 15 de mayo.
Ayudantía:	Viernes 15 de mayo.
Entrega:	**Jueves 28 de mayo a las 23:59 horas**.

En este laboratorio estudiaremos el funcionamiento de los operadores físicos en sistemas de bases de datos de grafos. Estos operadores se encargan de manipular datos almacenados en la base de datos de acuerdo con la consulta que el usuario envía al sistema. En esta tarea usted tendrá la posibilidad de implementar el operador block-nested-loop join y un operador de materialización. Por último, como tarea optativa (bonus) usted podrá implementar los operadores de external-merge sort y sort-merge join.

Bindings y sus operadores

La semántica del lenguaje de consultas de IIC3413-DB esta basado en "bindings" en vez de tuplas. Para entender este concepto considere la siguiente consulta en IIC3413-DB :

```
SELECT ?n.name
MATCH (?n :Person) -[:knows]-> (?m :Person)
WHERE NOT ?n.name == ?m.name
```

Como se puede ver, el lenguaje de consultas de una base de datos de grafos hace uso de variables para hacer referencia a objetos. Por ejemplo, `?n` y `?m` representan dos nodos del grafo, que después son usados para comparar sus nombres al verificar `?n.name == ?m.name`. Dado lo anterior, es natural utilizar "bindings" en una base de datos de grafos para codificar cada output, en vez de tuplas. Formalmente, sea V un conjunto de variables y D un conjunto de objetos o datos (por ejemplo, nodos de un grafo). Un binding $b : V \rightharpoonup D$ es una función parcial que asigna variables en V a objetos en D. Por ejemplo, si $V = \{\texttt{?n}, \texttt{?m}, \texttt{?r}\}$ y D son los nodos del grafo, entonces algunos ejemplos de bindings pueden ser $b_1 = \{\texttt{?n} \to v_1, \texttt{?m} \to v_2\}$ y $b_2 = \{\texttt{?n} \to v_1\}$ donde v_1 y v_2 son dos nodos cualquiera del grafo.

Es importante notar que un binding es una función parcial por lo que no necesariamente todas las variables tienen que estar asignadas. En particular, en b_1 y b_2 no todas las variables están asignadas. Notar también que una tupla es un caso particular de un binding. En palabras simples, una tupla t de una relación R es una asignación de los atributos de R a valores (esto es, objetos). Si consideramos los nombres de atributos como variables, entonces una tupla vendría siendo el caso especial de un binding, donde todas las variables, es decir atributos de R, están asignadas.

Una consulta Q de IIC3413-DB recibe un grafo G y su output $Q(G)$ es un conjunto de bindings sobre las variables en Q. Estas consultas se construyen en base a operaciones sobre conjuntos de bindings, similar al algebra relacional. De hecho, es fácil extender las operaciones de algebra relacional a conjunto de bindings. En este laboratorio trabajaremos solo con el operador join y de ordenamiento de bindings, así que solo definiremos estos operadores formalmente (otros operadores, como selección y proyección, también se pueden definir).

Sea V un conjunto de variables y D el conjunto de datos. Para un binding $b : V \rightharpoonup D$ se define como $\text{dom}(b)$ las variables asignadas por b. Por ejemplo, $\text{dom}(b_1) = \{\texttt{?n}, \texttt{?m}\}$ y $\text{dom}(b_2) = \{\texttt{?n}\}$. Para dos bindings b y b' decimos que b y b' son compatibles, denotado por $b \sim b'$, si para todo $\texttt{?x} \in \text{dom}(b) \cap \text{dom}(b')$ se tiene que $b(\texttt{?x}) = b'(\texttt{?x})$. Por ejemplo, es fácil ver que $b_1 \sim b_2$ y $b_1 \sim b_3$ donde $b_3 = \{\texttt{?m} \to v_2, \texttt{?r} \to v_3\}$. En particular, $b_2 \sim b_3$ ya que no comparten variables en común. Ahora, si consideramos el binding $b_4 = \{\texttt{?n} \to v_2, \texttt{?r} \to v_3\}$ es fácil ver que b_1 no es compatible con b_4 porque $\text{dom}(b_1) \cap \text{dom}(b_4) = \{\texttt{?n}\}$ pero $b_1(\texttt{?n}) \neq b_4(\texttt{?n})$.

Ahora, si tenemos dos bindings compatibles $b \sim b'$ podemos definir la unión de ellos. Específicamente, la unión de b y b', denotado por $b \cup b'$, cumple que $\text{dom}(b \cup b') = \text{dom}(b) \cup \text{dom}(b')$ y para toda variable $\texttt{?x}$, si $\texttt{?x} \in \text{dom}(b)$, entonces $b \cup b'(\texttt{?x}) = b(\texttt{?x})$, y $b \cup b'(\texttt{?x}) = b'(\texttt{?x})$, en otro caso. En otras palabras, $b \cup b'$ es la unión de las funciones parciales como si fueran una "relación". Volviendo a nuestros ejemplos tenemos que $b_1 \cup b_2 = \{\texttt{?n} \to v_1, \texttt{?m} \to v_2\}$, $b_1 \cup b_3 = \{\texttt{?n} \to v_1, \texttt{?m} \to v_2, \texttt{?r} \to v_3\}$ y $b_2 \cup b_3 = b_1 \cup b_3$. Notar que cuando dos bindings son compatibles, las variables que coinciden no hacen problemas al unirlos ya que contienen los mismos valores.

Para dos conjuntos de bindings B y B', se define el join $B \bowtie B'$ como:

$$B \bowtie B' \;=\; \{\, b \cup b' \;\mid\; b \in B, b' \in B' \text{ y } b \sim b' \,\}.$$

Esto es, juntamos cada binding de B con cada binding de B' que sean compatibles. Por ejemplo, si definimos $B_1 = \{b_1, b_2\}$ y $B_2 = \{b_3, b_4\}$, entonces tenemos que $B_1 \bowtie B_2 = \{b_1 \cup b_3, b_2 \cup b_3\}$. Intuitivamente, es fácil ver que el join entre conjuntos de bindings que acabamos de definir es la extensión del natural join en algebra relacional.

Por último, podemos definir el operador de sorting sobre bindings de la siguiente manera. Sea $<$ un orden total sobre el conjunto de datos D. Sea $V = \{\texttt{?x1}, \texttt{?x2}, \ldots, \texttt{?xn}\}$ y algún orden en particular (acá escogimos $\texttt{?x1} < \texttt{?x2} < \ldots < \texttt{?xn}$). Para dos bindings b y b' decimos que $b < b'$ si, y solo si:

1. $b(\texttt{?x1}) < b'(\texttt{?x1})$ cuando $\texttt{?x1} \in \text{dom}(b) \cap \text{dom}(b')$,
2. $\texttt{?x1} \in \text{dom}(b)$ cuando $\texttt{?x1} \notin \text{dom}(b')$, o
3. $b < b'$ usando el orden $\texttt{?x2} < \ldots < \texttt{?xn}$ cuando $\texttt{?x1} \notin \text{dom}(b)$ y $\texttt{?x1} \notin \text{dom}(b')$.

En otras palabras, ordenamos $b < b'$ por el valor de la primera variable $\texttt{?x1}$ si esta definida por lo menos en b. En cambio, si tanto b como b' no contienen $\texttt{?x1}$ entonces miramos la segunda variable y así recursivamente. Es fácil ver que este orden define un orden total sobre bindings, y nos permite definir el operador sorting sobre un conjunto de bindings B.

La representación de bindings en IIC3413-DB

En IIC3413-DB , las estructuras de datos que representan los "bindings" son la clase `Binding` (en *base/binding*) y la clase `BindingId` (en *relational_model/binding*). La primera es una clase que esconde un diccionario que mapea variables, representadas por la clase `VarId` (en *base/ids*) a objetos de la base de datos, representada por la clase `GraphObject` (en *base/graph*). En estricto rigor la clase `Binding` es una interfaz y, por ejemplo, una implementación de ella es `BindingMatch` (en *relational_model/binding*).

La segunda clase que se encarga de representar un binding, y la más importante para este laboratorio, es la clase `BindingId`. Esta clase es un caso especial de binding, donde cada variable es mapeada a un id de un objeto, implementado por la clase `ObjectId` (en *base/ids*). Con el objetivo de hacer los operadores lo más eficientes posibles, a bajo nivel IIC3413-DB representa cada objeto (como nodos, aristas, o incluso strings) con un identificador. De esta manera, las operaciones de joins se pueden hacer más eficientes simplificando, por ejemplo, el ordenamiento y comparación de objetos. Por este motivo, `BindingId` representa bindings que mapean variables a ids.

En el caso de `BindingId`, su implementación es más eficiente que la de `Binding`, ya que se usa un arreglo `dict` (de hecho, un `vector` en C++) para representar la función parcial entre variables y ids. Para esto,

cuando IIC3413-DB recibe una consulta Q e identifica su conjunto de variables $V = \{\texttt{?x1}, \texttt{?x2}, \ldots, \texttt{?xn}\}$, este las ordena y asigna un número del 1 al n. De ahí, `BindingId` codifica un binding b de V tal que si $\texttt{?xi} \in \text{dom}(b)$, entonces $\texttt{dict}[i]$ contiene al valor $b(\texttt{?xi})$. En cambio, si $\texttt{?xi} \notin \text{dom}(b)$, entonces el valor en $\texttt{dict}[i]$ no esta iniciado (es igual a `null`). En esta codificación, IIC3413-DB esta sacrificando espacio por tiempo, ya que si bien el arreglo de `BindingId` tiene espacio asignado para variables que no contiene, el acceso a cada variable que si contiene es mucho más eficiente.

Implementación de operadores físicos

Para implementar los operadores físicos, IIC3413-DB provee la interfaz `BindingIter` (en *base/binding*) y la interfaz `BindingIdIter` (en *relational_model/binding*) para iterar por conjuntos de bindings del tipo `Binding` o `BindingId`, respectivamente. En este laboratorio trabajaremos solo sobre operadores físicos que operan con `BindingId`, por lo que de ahora en adelante solo explicaremos los detalles de `BindingIdIter`.

Todo operador que implementa la interfaz `BindingIdIter` debe implementar los siguientes métodos:

```
void begin(BindingId& input);
void reset(BindingId& input);
BindingId* next();
```

La función de `begin` esta encargada de iniciar el operador físico y `reset` de reiniciarlo. Ambas funciones reciben como input un BindingId que puede ser útil para su inicialización. En cambio, la función `next` entrega el siguiente binding y `null` si se llego al final de todos los bindings. En particular, esta interfaz no tiene un método de cerrar ("close"), como se vio en clases para el caso de una base de datos relacional.

Un operador físico sobre `BindingId` implementado en IIC3413-DB es el index-nested-loop join, codificado por la clase `IndexNestedLoopJoin` (en *relational_model/physical_plan/binding_id_iter*). Esta clase implementa la interfaz `BindingIdIter` y es un buen ejemplo para entender el funcionamiento de un operador físico en IIC3413-DB. En este caso, el input que recibe `IndexNestedLoopJoin` en sus métodos de `begin` y `reset` es el siguiente binding que se desea buscar, y `next` itera sobre los siguientes resultados que hacen "match" con ese binding. Otro operador físico sobre `BindingId` es un scan completo sobre un grupo de objetos, implementado por la clase `TotalScan` (en *relational_model/physical_plan/binding_id_iter*). En este caso, `begin` y `reset` inician el scan por los objetos y `next` entrega el siguiente binding. Para esta clase, los inputs recibidos por `begin` y `reset` no son tan relevantes, pero necesarios para definir el número de variables de los `BindingId`.

Para entender y probar el funcionamiento de varios operadores físicos conectados, esto es, un "plan físico", se recomienda ejecutar IIC3413-DB, desde los archivos *lab3_example.cc* o *lab3_example2.cc* (en carpeta *main*). Aquí se muestran ejemplos de como levantar el motor, escribir un plan físico desde código y ejecutarlo.

Tarea 1: Implementación de block-nested-loop join (3 puntos)

Para esta primera tarea del laboratorio usted debe implementar el algoritmo de block-nested-loop join visto en clases, pero adaptado para el caso de bindings (según la semántica expuesta más arriba). Para esto, su clase llamada `BlockNestedLoopJoin` debe implementar la interfaz `BindingIdIter` y, en particular, debe tener los siguientes métodos:

```
BlockNestedLoopJoin(std::unique_ptr<BindingIdIter> left,
                    std::unique_ptr<BindingIdIter> right, int buffer_size);
~BlockNestedLoopJoin();
void begin(BindingId& input);
void reset(BindingId& input);
BindingId* next();
```

Como se puede ver en la definición anterior, similar al operador `IndexNestedLoopJoin` este debe recibir como parámetro los iteradores hijos de la izquierda y de la derecha, como también la cantidad de buffer que

ocupara su operador. Para simplificar la implementación de este operador, el buffer ocupado será un arreglo dínamico de `BindingId` de tamaño `buffer_size`. En otras palabras, el tamaño del buffer se definirá al crear el operador y usted debe simular este buffer internamente con un arreglo. En particular, no es necesario utilizar el buffer manager del sistema.

Tarea 2: Operación de materialización (3 puntos)

Para esta segunda tarea, usted debe implementar un operador físico de "materialización" de resultados. En otras palabras, su operador llamado `Materialize` debe implementar la interfaz `BindingIdIter` y tener los siguientes métodos:

```
Materialize(std::unique_ptr<BindingIdIter> left);
~Materialize();
void begin(BindingId& input);
void reset(BindingId& input);
BindingId* next();
```

Este operador itera sobre los `BindingId` en `left` y los materializa en disco para ser usados una segunda vez que sean llamados. Esto es, la primera vez que se llama el operador y se hace `next()`, este operador pide el siguiente binding a `left`, lo guarda en un archivo temporal y lo entrega como resultado. Asi se continua por cada nuevo `next`. Una vez que el operador se llama por segunda vez llamando a `reset` este no pide a `left` los binding, si no que los extrae directamente del archivo donde los materializó.

Para la implementación de este operador usted debe ocupar las clases `FileManager` y `BufferManager` utilizadas en los laboratorios anteriores. En particular, el `FileManager` de la nueva versión de IIC3413-DB lo provee con un método `FileId get_tmp_file_id()` que le permite crear un archivo temporal donde, a través del `BufferManager`, usted podrá acceder para almacenar sus bindings. Es importante recordar que `BufferManager` provee un acceso de páginas al disco y usted debe almacenar y codificar los bindings (que son tuplas de tamaño fijo) en cada página para después accederlos. Para esto, usted puede ocupar la codificación que encuentre más conveniente para su solución.

Bonus: Operación de external sorting (1 punto)

Para esta tarea usted debe implementar el operador físico de external-merge sort. La clase que implementa este operador debe llevar por nombre `ExternalMergeSort`, debe implementar la interfaz `BindingIdIter` y tener los siguientes métodos:

```
ExternalMergeSort(std::unique_ptr<BindingIdIter> left);
~ExternalMergeSort();
void begin(BindingId& input);
void reset(BindingId& input);
BindingId* next();
```

Como vimos en clases, este operador debe recibir un `BindingIdIter` `left` y entregar los bindings ordenados según el orden de bidings definido al comienzo del enunciado. El orden de las variables a utilizar debe ser el definido de manera global por el arreglo `dict` (o sea, empezar por el valor en la posición 1 del arreglo, etc). Para esta tarea, usted debe utilizar el `BufferManager` para hacer el ordenamiento externo y se le recomienda utilizar la experiencia ganada en el ejercicio anterior de este laboratorio (no necesariamente debe utilizar el operador `Materialize` directamente).

Bonus: Operación de sort-merge join (1 punto)

Para esta última tarea usted debe implementar el operador físico de sort-merge join. La clase que implementa este operador debe llevar por nombre `SortMergeJoin`, debe implementar la interfaz `BindingIdIter` y tener los siguientes métodos:

```
SortMergeJoin(std::unique_ptr<BindingIdIter> left,
              std::unique_ptr<BindingIdIter> right);
~SortMergeJoin();
void begin(BindingId& input);
void reset(BindingId& input);
BindingId* next();
```

Este operador debe realizar el join de `left` y `right` utilizando el algoritmo de sort-merge join visto en clases. Para esto, usted puede ocupar la implementación del external-merge sorting del ejercicio anterior.

Bonus por bugs encontrados/solucionados en IIC3413-DB

Se darán los siguientes bonus por encontrar o solucionar un bug de IIC3413-DB :

1. **Bug encontrado**: se darán 2 décimas en la nota final del laboratorio si encuentran un bug. Para ser efectivo este bonus, deben publicar una issue[1] del repositorio del proyecto sobre el bug, esto es, la descripción del bug, junto con el input que se le está entregando al sistema más el output "errado" que entrega.

2. **Bug solucionado**: se darán 2 décimas adicionales por entregar el código que soluciona este bug en el sistema (no necesariamente tiene que ser el mismo estudiante). Para esto deben dar una explicación de como solucionar el problema en el mismo issue y hacer un pull request con el código que soluciona el problema (no importa si es una línea o muchas líneas). Aparte de tener las dos décimas, su usuario GitHub quedará como contribuidor al proyecto.

Una vez publicado el bug o solución, los ingenieros del proyecto revisarán la publicación y determinarán si el bonus corresponde o no. Por último, dado un bug o solución se dará el bonus al primer estudiante que lo publiqué en las issues del repositorio.

Nueva versión de IIC3413-DB

Para este laboratorio, usted debe utilizar la nueva versión de IIC3413-DB (branch Lab3) en el repositorio del proyecto, para la cual se arreglaron algunos errores y se extendieron algunas funcionalidades para esta entrega. Para esta entrega usted debe utilizar la versión nueva de IIC3413-DB y, en particular, no puede utilizar las versiones usadas en los laboratorio anteriores.

Evaluación y entrega

El día límite para la entrega de esta tarea será el Jueves 28 de mayo a las 23:59 horas. Para ello se utilizará el repositorio privado en GitHub que fue proporcionado por el curso. Para la entrega usted deberá crear una nueva rama en git a partir de la rama principal del proyecto con nombre "laboratorio3", para que así pueda seguir desarrollando sus próximos laboratorios sin considerar las modificaciones del actual. La evaluación se realizará en base a test que debe pasar su código.

Ayudantía y preguntas

El día Viernes 15 de mayo se realizará una ayudantía donde se darán más detalles sobre IIC3413-DB y el laboratorio. Para preguntas se pide usar el foro del curso o enviar un correo a `iic3413@inc.puc.cl`. De preferencia, se sugiere escribir al foro del curso para que todos los estudiantes estén al tanto de las dudas.

[1] https://github.com/PUC-IIC3141/IIC3413-DB/issues