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 4

Laboratorio:	Optimización de consultas.
Publicación:	Viernes 12 de junio.
Ayudantía:	Viernes 12 de junio.
Entrega:	**Viernes 26 de junio a las 23:59 horas**.

En este laboratorio usted estudiará el funcionamiento del optimizador de consultas de IIC3413-DB e implementará el optimizador de Selinger para encontrar el plan óptimo para un conjunto de joins.

Planes lógicos en IIC3413-DB

Al igual que cualquier sistema de bases de datos relacional, IIC3413-DB crea un plan lógico desde una consulta, construyendo un primer plan para después transformarlo en un plan físico. El plan lógico se construye a partir del resultado del módulo de parsing que se encuentra en `base/parser`. Aquí el encargado de representar un plan lógico es la clase `Op`, ubicada en `base/parser/logical_plan/op`. Cada operación lógica en IIC3413-DB es representada como una extensión de la clase `Op`. Por ejemplo, la clase `OpMatch` es el operador lógico encargado del `MATCH` en una consulta. Por lo tanto, la composición de los objetos que extienden la clase `Op` forman el plan lógico de la consulta.

Para construir el plan lógico, la clase `Op` cuenta con dos método estáticos encargados de esta responsabilidad:

```
static std::unique_ptr<OpSelect> get_select_plan(std::string query);
static std::unique_ptr<OpSelect> get_select_plan(ast::Root& ast);
```

El método `get_select_plan` con input `std::string query`, recibe la consulta y llama al módulo de parsing[1]. Este módulo construye un árbol de parsing, representado por un objeto del tipo `ast::Root`, para después llamar al método `get_select_plan` con input `ast::Root ast` para construir el plan lógico representado por un objeto de la clase `OpSelect`. Al final de este proceso, toda la información de la consulta queda almacenada en este objeto que representa el plan lógico.

Optimizador de consultas en IIC3413-DB

El encargado de construir el plan físico y optimizar la consulta es la clase `QueryOptimizer`, ubicada en `relational_model/query_optimizer`. Esta clase implementa la interfaz `OpVisitor`, la cual contiene los métodos para "visitar" un plan lógico, esto es, para visitar los objetos de la clase `Op`. Especificamente, al llamar el método:

```
std::unique_ptr<BindingIter> exec(OpSelect&);
```

de `QueryOptimizer` con el objeto `OpSelect` que representa al plan lógico, esté recorre los objetos de tipo `Op` para construir el plan físico de la consulta. Para esto, `QueryOptimizer` implementa cada uno de los métodos

[1]Para este laboratorio, no es necesario que entienda el funcionamiento del módulo de parsing, más allá de los explicado en este enunciado.

de `OpVisitor`, encargados de visitar cada una de las extensiones de `Op`. Por ejemplo, al implementar el método `visit(OpMatch&)` la clase `QueryOptimizer` se encarga de visitar `OpMatch` y construir el plan físico correspondiente a la información contenida en el operador lógico para la clausula `MATCH` de la consulta (ver implementación de este método en `query_optimizer.cc` para más información).

A diferencia de lo estudiado en clases, en IIC3413-DB el plan físico de una consulta no esta representado exclusivamente por una composición de operadores físicos (vistos en el laboratorio anterior). Para mantener un desacoplamiento entre los operadores físicos de joins y las labores de estimación de cardinalidad, es que el plan físico para `MATCH` está representado por una composición de objetos de la clase `JoinPlan`, ubicadas en `relational_model/query_optimizer/join_plan`. Aquí el plan físico de `MATCH` es la composición de objetos del tipo `JoinPlan`, donde cada uno de ellos representa un posible operador físico junto las responsabilidades de estimar el costo del operador, cardinalidad, variables que contiene y otras. Por ejemplo, los objetos `JoinPlan` también cuentan un método `print` que permite imprimir el plan en consola y que seguramente le ayudará para el debugging de su solución.

Optimización del orden de joins

Al construir el plan físico, el `QueryOptimizer` también se encarga de decidir el orden de los joins de la clausula `MATCH`, esto es, decidir el orden como están compuestos los objetos `JoinPlan`. Esta responsabilidad está actualmente encapsulada en `QueryOptimizer`, específicamente, en el método:

```
std::unique_ptr<BindingIdIter>
              get_greedy_join_plan(std::vector<std::unique_ptr<JoinPlan>>& base_plans);
```

Este método recibe una lista de planes básicos y construye un plan de joins siguiendo una estrategia *greedy*: en cada iteración escoge la composición (join) del plan actual con un "base plan" que, una vez combinados, tengan el menor costo estimado. En esta decisión, el método incluso escoge si realizará un index nested loop join o un sort-merge join entre el plan actual y un base plan, como también trata en lo posible de evitar hacer un producto cruz. Para estimar cual es el costo de cada plan, los distintos `JoinPlan` cuentan con la función `estimate_cost()` y `estimate_output_size()` que entregan una estimación del costo y del tamaño del output del plan, respectivamente. Para más detalles, se recomienda revisar la implementación del método `get_greedy_join_plan` en `query_optimizer.cc`.

Por último, es importante notar que el método `get_greedy_join_plan` es llamado unicamente al final del método `visit(OpMatch& op_match)` donde, después de visitar el objeto `OpMatch`, se decide el orden óptimo de los joins.

Tarea: Implementación de optimizador de Selinger (6 puntos)

Para esta tarea usted debe extender IIC3413-DB implementando el optimizador de Selinger para decidir el orden óptimo de los joins de los base plans. Para esto, usted debe crear un nuevo método en `QueryOptimizer` llamado:

```
std::unique_ptr<BindingIdIter>
             get_selinger_join_plan(std::vector<std::unique_ptr<JoinPlan>>& base_plans);
```

y entregar el plan óptimo según la estrategia de Selinger vista en clases. Para hacer uso de este nuevo plan, usted debe modificar el método `visit(OpMatch& op_match)` y, en vez de llamar a `get_greedy_join_plan`, llamar su nuevo método para el optimizador de Selinger.

Para simplificar su solución, usted puede restringir su búsqueda de planes únicamente al uso de index nested loop join, representado por la clase `NestedLoopPlan` (en otras palabras, no necesita también decidir si usará un `MergePlan` como en la implementación greedy). También, usted debe hacer uso de las funciones de estimación de costo de `NestedLoopPlan`, aunque estás no son necesariamente óptimas.

Bonus: Mejorar la estimación de cardinalidad de `NestedLoopPlan` (1 punto)

Actualmente la estimación de cardinalidad, esto es, el método `estimate_output_size` de `NestedLoopPlan`, es muy ingenuo, estimando la cardinalidad de un join como la multiplicación entre las cardinalidades. Para este bonus, usted debe mejorar la estimación de cardinalidad de este método según las técnicas vistas en clases. Para esto, usted puede ocupar los métodos del catálogo de IIC3413-DB (o extenderlo) que se encuentran en la clase `Catalog`, ubicada en `storage/catalog`.

Para este bonus, la evaluación no se hará con test unitarios. Debido a esto, usted debe implementar su mejora y documentar en el método la estrategia implementada.

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[2] 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 Lab4) 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 Viernes 26 de junio 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 "laboratorio4", 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 12 de junio 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.

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