Описание менеджера эвристик

Введём некоторые пояснения.
Будем измерять сложность компоненты по одной из двух метрик. На её основе введём функцию бюджета в компоненте. Если его достаточно, то можно позволить более долгое время оптимизации и более оптимальные эвристики/алгоритмы. Метрики:

1. Количество связных подграфов (#csg). Это соответствует числу записей в ДП таблице.
2. Количество пар комплементарных связных подграфов (#ccp). Пусть S_1 и S_2 непересекающиеся подмножества отношений в графе, также S_1 и S_2 - связанные подграфы. И существует ребро между S_1 и S_2. Тогда (S_1, S_2) это пара комплементарных связных подграфов (csg-cmp-pair ccp).

Каждая эвристика/алгоритм в зависимости от топологии и её сложности, подаваемых на вход, будет иметь стоимость. Эта стоимость будет позже вычитаться из бюджета связной компоненты после планирования.
Пусть **border_chain**, **border_cycle**, **border_star**, **border_density_graph**, **border_tree**, **border_mixed** - это количество связных подграфов, больше которого топология считается сложной, меньше - лёгкой.
Звезда это центральная вершина и связанные с ней цепи, каждые две цепи не связаны друг другом. Вершина будет центральной, если выполнено одно из двух:

1. Вершина имеет не меньше 2 соседей и количество её строк более чем в **border_star_dominance** раз больше чем у каждого из соседей.
2. Соседей не меньше 3.

Назовём граф плотным, если количество рёбер в нём не меньше количества рёбер в плотном графе (от такого же числа вершин), умноженного на **border_density**.

Первый подход:

Пусть дан большой аналитический запрос. Для его планирования будем выполнять следующие шаги:

1. Представим запрос в виде набора связных компонент, любые две таблицы из разных компонент не имеют условий соединения между собой.
2. Выделим каждой компоненте бюджет.
3. Для каждой связной компоненты будем выполнять итеративное разбиение на топологии и их планирование в цикле, пока не получим один план для компоненты. Если что-то получилось выделить, возвращаемся в начало цикла, если нет, то переходим на планирование выделенных топологий и снова в начало цикла:
   (a) Выделим из компоненты **плотные графы** (с 4+ вершинами).
   (b) Выделим циклы.
   (c) Если после выделения плотных графов и циклов, компонента остаётся связной, то выделим дерево, иначе выделяем звёзды и цепи.
   (d) Выделим звёзды с лучами длины до **ray_length**.
   (e) Выделим цепи оставшиеся цепи.

Распределим бюджет компоненты по топологиям в соответствии с их сложностями. Каждая топология после планирования становится вершиной, получаем новую связную компоненту меньшего размера.
Планирование:

(a) Если бюджета для текущей топологии не осталось, то планируем эвристически, иначе по сложности: для лёгкой топологии используем встроенный ДП, для тяжёлой эвристику. Эвристики для топологий:

i. Цепь:
   A. Дешёвая эвристика: итеративно будем выполнять шаг алгоритма GOO пока стоимость соединения не помещается в бюджет или сложность не станет лёгкой, затем передадим встроенному ДП, с сохранением результатов.
   B. Дорогая эвристика: используем ДП алгоритм для цепочки (описан ниже).

ii. Цикл: найдем соединение, результат которого имеет наибольшую кардинальность, и разобьём цикл по этом соединению - уберём одну таблицу в этом соединении. Спланируем цепь, и присоединим удалённую таблицу.

iii. Звезда:
   A. Сначала спланируем лучи как цепи, затем будем последовательно присоединять к центру полученные отношения, жадно по кардинальности.
   B. Последовательно, жадно по кардинальности, присоединяем к центру начала соседних цепочек.

iv. Плотный граф:
   A. Лёгкая эвристика: используем GOO.
   B. Дорогая эвристика: IDP-2. Используем GOO, не делая соединения, получаем порядок в виде bushy дерева, затем выбираем самое дешёвое дерево размера до **k**, планируем его и повторяем, пока не останется одно отношение.

v. Дерево:

   Дешёвая эвристика GOO.

   Дорогая эвристика IKKBZ([TI84][RK86])/linDP++.

После планирования топологий на текущей итерации вычтем из бюджета компоненты потраченную сумму.
Делаем проверку на выполнение условий для эвристик на OUTER JOINS и гипер-рёбра (описаны ниже). Если условие выполнено, добавим соответствующее ребро между сторонами. Тогда возможны два случая:

Ребро было добавлено внутри одной связной компоненты - ничего не меняется.

Ребро было добавлено между двумя разными связанными компонентами - будет новая связная компонента, у которой бюджет это сумма бюджетов двух родительских компонент. Заменим две компоненты на новую.

4. Объединим планы компоненты запроса алгоритмом GOO(по стоимости) с помощью декартового произведения.

Второй подход на планирование деревьев:

1. Представим запрос в виде набора связных компонент, любые две таблицы из разных компонент не имеют условий соединения между собой.

2. Выделим каждой компоненте бюджет.

3. Для каждой связной компоненты выполним два шага и получим план для компоненты.
4. Первый шаг

   Выделим минимальное по стоимости или кардинальности остовное дерево.

   Останутся одиночные отношения, добавление которых в дерево добавило бы рёбра приводящие к циклам.

   Бюджет компоненты используется для планирования дерева. После получим новую связную компоненту меньшего размера в виде топологии дерева или звезды. В зависимости от оставшегося бюджета и сложности новой топологии выполним планирование нового дерева или звезды.

   После каждого шага вычтем из бюджета компоненты потраченную сумму. Выполняем проверку и обработку эвристик на гипер-графы и OUTER JOINS после каждого шага как в первом подходе.
5. Объединим планы компоненты запроса алгоритмом GOO(по стоимости) с помощью декартового произведения.

Гипер-рёбра INNER JOINS:

Если условие соединяет более двух таблиц, представляем его как гипер-ребро. Пусть условие охватывает таблицы $R_1, \ldots, R_n$. Тогда добавим данное ребро в граф тогда, когда каждое R_i принадлежит одному из отношений A или B, A и B отличны.

LEFT OUTER JOINS:

При генерации плана сначала выполняем соединения всех таблиц левой стороны между собой (и с другими таблицами, не связанными с правой стороной, но связанными с левой). После присоединяем правую часть.

RIGHT OUTER JOINS:

Аналогично LEFT OUTER JOINS, только наоборот.

FULL OUTER JOINS:

Соединим левую сторону с теми отношениями, у которых есть связь с левой стороной и отсутствует с правой. Аналогичное сделаем и для правой стороны. Получится для независимых, по включению таблиц, отношения.

Третий подход на основе статей про DPHyp[GM] и linDP++[BR] и адаптивный фреймворк[TN].

1. Представим запрос в виде набора связных компонент, любые две таблицы из разных компонент не имеют условий соединения между собой.
2. Выделим каждой компоненте бюджет.
3. Для каждой связной компоненты выполним и получи план компоненты:

   Если компонента лёгкая (**border_mixed**) и бюджета достаточно, то планируем компоненту целиком с помощью DPHyp.

   Иначе в зависимости от сложности и бюджета используем одну из эвристик:

Тяжёлая эвристика: IKKBZ([TI84][RK86])/linDP++.

Лёгкая эвристика: IDP-2 с DPHyp в качестве ДП планировщика.

Бюджет компоненты используется для планирования дерева. После получим новую связную компоненту меньшего размера в виде топологии дерева или звезды. В зависимости от оставшегося бюджета и сложности новой топологии выполним планирование нового дерева или звезды.

После каждого шага вычтем из бюджета компоненты потраченную сумму.

4. Объединим планы компоненты запроса алгоритмом GOO(по стоимости) с помощью декартового произведения.

Описание некоторых алгоритмов эвристик:

1. GOO [Moe25]: эвристика для создания bushy деревьев. Соединяет пары, те пары отношений, которые имеют наименьшую стоимость (кардинальность) объединения. Повторяем, пока не останется одно отношение.

2. IDP-2 [DK]: сначала применяется жадная эвристика для построения деревьев соединений размером до k. Полученное поддерево динамически планируется. Полученное отношение заменяет поддерево в дереве. Повторяем итерации пока не получим одно отношение.

3. ДП алгоритм для цепочки:

   Создаём DP-таблица dp[i][j], где i и j — индексы начала и конца подцепочки ($1 \leq i \leq j \leq n$).

   dp[i][j] хранит: план для отношения с i по j и точку разбиения k (делит [i, j] и минимизирует стоимость).

   Для всех i : dp[i][i] =(R_i, -1)

   Для $i \neq j$: dp[i][j] = $\inf$

   В тройном цикле перебираем k: для l = 2 . . . n, для i = 1 . . . n - l + 1, j = i + l - 1, для k = i . . . j - 1

   Выбираем такой k, где стоимость соединения подцепочек [i, k] и [k+1, j] минимальна. dp[i][j] =([i, k] $\bowtie$ [k + 1, j], k)

   Результат в dp[1,n].

Список литературы

[BR] Thomas Neumann Bernhard Radke. Lindp++: Generalizing linearized dp to crossproducts and non-inner joins. *Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018.*

[DK] Konrad Stocker Donald Kossmann. Iterative dynamic programming: A new class of query optimization algorithms.

[GM] Thomas Neumann Guido Moerkotte. Dynamic programming strikes back. *SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.*

[Moe25] Guido Moerkotte. Building query compilers. pages 48 – 49, 31.01.2025.

[RK86] Carlo Zaniolo Ravi Krishnamurthy, Haran Boral. Optimization of nonrecursive queries. *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 128–137, 1986.

[TI84] Tiko Kameda Toshihide Ibaraki. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS), Volume 9, Issue 3*, pages 482 – 502, 1984.

[TN] Bernhard Radke Thomas Neumann. Adaptive optimization of very large join queries. *In Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD'18), 16 pages.*