МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет — Механико-математический

Кафедра — Программирования
Направление подготовки — Математика и компьютерные науки

РЕФЕРАТ

Пьянзин Богдан Олегович

(Фамилия, Имя, Отчество автора)

Тема: Методы адаптивного планирования запросов в реляционных СУБД

«Реферат принят»

Научный руководитель

к.ф-м.н., с.н.с.,
старший преподаватель
кафедры программирования ММФ

Пономарёв Д.К. / _____
(ФИО, Подпись)

«...» ..... 2025 г.

Новосибирск, 2025

Оглавление

Введение

Актуальность темы

Современные СУБД работают с большим объёмом информации и транзакций, используя для ввода запросов язык SQL. В процессе трансляции запрос превращается сначала в логическое представление в виде дерева, затем с помощью оптимизатора СУБД в физический план исполнения. Производительность СУБД напрямую зависит от качества построенного физического плана. Для создания "хорошего" плана нужно решить или приблизить решение NP полной задачи перебора соединений таблиц, так как оно требует сложных вычислений и значительных затрат ресурсов.
Сложность задачи обусловлена тем, что количество возможных соединений(бинарных) с n таблицами равно количеству бинарных деревьев с n листьями - Числа Каталана, которые имеют экспоненциальную скорость роста. В классических подходах используется динамическое программирование(DP) и эвристический поиск для решения этой проблемы. Однако с ростом количества таблиц, потребление памяти(для хранения промежуточных результатов) и времени планирования становятся слишком велики. Эвристические методы не гарантируют глобальную оптимальность решения, так как находят локально оптимальные планы.
Таким образом, выбор неэффективного плана запроса может привести к значительному замедлению работы системы, а процесс поиска хорошего решения является нетривиальной задачей. В свою очередь оптимальный план позволяет эффективно исполнить запрос, с приемлемым потреблением CPU, памяти, I/O, сетевых ресурсов(особенно актуально для распределённых СУБД).

Цель

Цель данного исследования – выявить факторы, влияющие на выбор порядка соединений в запросах к СУБД, а также рассмотреть классические подходы, применяемые для оптимизации данной задачи. В рамках работы планируется:

- Определить ключевые параметры, влияющие на производительность операций соединения (размер таблиц, наличие индексов, тип соединения и др.).
- Разобрать механизмы планирования запросов в **PostgreSQL**.
- Рассмотреть традиционные методы планирования порядка соединений, включая динамическое программирование, эвристические алгоритмы и генетические, используемые в **PostgreSQL**, изучить их эффективность и ограничения при увеличении числа соединений в запросе.
- Выработать критерии для выбора оптимального метода планирования соединений в зависимости от типа запроса и структуры данных.

Исследование позволит оценить, какие подходы обеспечивают наилучшую производительность SQL-запросов в различных условиях и предложить рекомендации по их применению.

Основная часть

Теоретические основы соединений таблиц в SQL запросах

Определение и классификация соединений в планировании запроса

Соединение таблиц – это операция, которая позволяет объединять данные из двух и более таблиц по определённому условию. Использование соединений необходимо, когда информация распределена между несколькими таблицами. В процессе работы оптимизатора, решается вопрос какой тип соединения использовать, как и в каком порядке соединить таблицы. Существуют три типа соединений:

Nested Loop Join – для каждой строки из левой таблицы выполняется проход по всем строкам правой таблицы и ищутся соответствующие строки по условию соединения. Пусть левая таблица занимает M страниц и имеет m строк, правая N, n соответственно. Имеется варианты NLJ:

- **Наивный NLJ** – перебирается каждая строка левой таблицы, сравнивается с каждой из правой. Сложность $O(M + (m * N))$. Работает медленно если таблицы большие, но прост в реализации, эффективен при малых таблицах.
- **Индексированный NLJ** – правая таблица имеет индекс по соединяемому полю. Сложность $O(M + (m * log(n)))$. Может значительно ускорить поиск, если правая таблица большая. Но будет всё ещё медленным если левая таблица большая.
- **Блочный поиск** – улучшенная версия наивного NLJ, при ограничении памяти. Пусть доступно B буферов. Загружаем B-2 блоков из левой таблицы, 1 буфер для правой таблицы, 1 под результат. Для каждой страницы из из буферов для левой таблицы, проверяем условие с каждой строкой из буфера правой таблицы, проходимся одним буфером по всей правой таблице. Сложность $O(M + [M/(B-2)] * N)$.

Merge Join – таблицы сортируются по ключу соединения, происходит построчное слияние таблиц. Очень хорошо работает если таблицы отсортированы и/или есть индекс по ключу, лучше чем NLJ на больших таблицах. Недостатки: потребность в сортировке, нужно память для хранения сортированных таблиц. Сложность $O($стоимость сортировки двух таблиц $+ M + N)$.

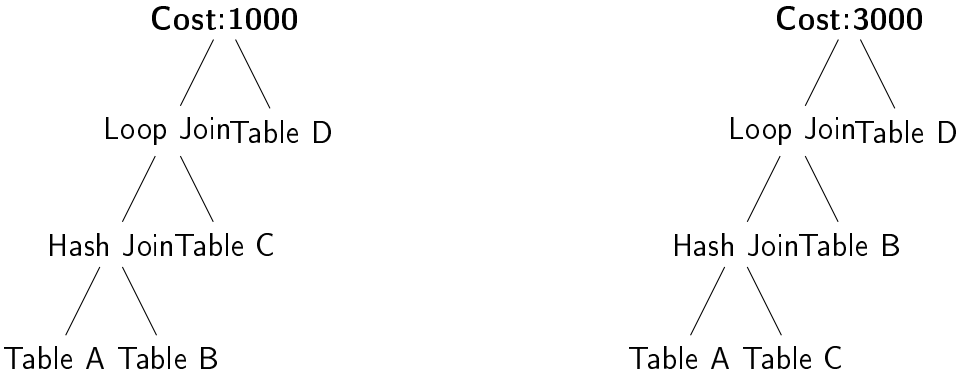
Hash Join – строится хэш-таблица для меньшей таблицы, перебирается каждая строка в большей таблице и проверяется соответствие в хэш таблице. Сложность $O(M + N)$. Работает хорошо если одна таблица большая другая маленькая и условие на равенство(=). Недостатки: требует память для хэш-таблицы, плохо работает если условие на есть диапазон.

Перечисленные выше способы соединения имеют разную эффективность, напрямую зависящую от статистик. Выбор неоптимального типа соединения может привести к значительному ухудшению стоимости исполнения плана. Определение нужного типа происходит в процессе оптимизации, то есть перевода логического дерева в физическое.

Задача выбора порядка соединений

С увеличением количества данных увеличивается и время, необходимое для выполнения запросов. Сокращение времени выполнения запросов становится важным фактором для

удобства и эффективности СУБД. Оптимизатор преобразует запрос в набор планов. Каждый план представлен в виде дерева, вершинами в которых являются данные из таблиц или результат соединения таблиц по условию, в общем случае называется отношением. Рёбра — условия соединения отношений. При этом операция соединения отношений не ассоциативна, то есть $(R1 \times R2) \times R3 \neq R1 \times (R2 \times R3)$ по стоимости. В статье [Liu24] приводится пример разной стоимости для двух деревьев соединений.



Выбор в какой последовательности нужно соединить таблицы является задачей выбора порядка соединений. Различные планы исполнения для одного и того же запроса возвращают одинаковый результат, но время и ресурсы (CPU, память, I/O обращения, возможно сетевые ресурсы), необходимые для выполнения запроса, сильно различаются. Поэтому выбор оптимального плана позволяет сократить время отклика, минимизировать потребление ресурсов и эффективно обрабатывать большие массивы данных, значительно повышая удобство работы пользователя.

Задача планировщика/оптимизатора — построить наилучший план выполнения. Если это не требует больших вычислений, оптимизатор запросов будет перебирать все возможные варианты планов, чтобы в итоге выбрать тот, который имеет наименьшую стоимость. Например, если для обрабатываемого отношения создан индекс, прочитать отношение можно двумя способами. Во-первых, можно выполнить простое последовательное сканирование, а во-вторых, можно использовать индекс, т.е есть выбор сканирования зависит от статистик. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый, который отдаётся исполнителю(ям).

Факторы влияющие на производительность

Исходя из классификации типов соединений при планировании запроса нужно иметь статистику: по размеру отношений, подаваемых на вход оператору, наличию индексов, наличие сортировки данных. Применение некоторых типов может потребовать дополнительной памяти. Помимо этого существуют следующие факторы:

Кардинальность отношения – количество отдельных строк в таблице базы данных. Если таблица большая, выполнение соединения с ней может быть дорогим, если же таблица маленькая, её можно поместить в память (in-memory), и не придётся обращаться к внешней памяти, что может повлиять на стоимость. плана.

Селективность соединения – количество строк, оставшихся после фильтрации. Наиболее селективные соединения следуют применять как можно раньше, так стоимость оператора соединения это стоимость дочерних операторов + собственная обработка.

Селективность столбцов – уникальность значений в столбце. Высокая селективность - много уникальных значений, низкая селективность - мало уникальных значений.

Корреляция столбцов – зависимость значений между столбцами. Если данные сильно коррелируют, оптимизатор может дать неправильную оценку оператору соединения. Тогда ошибка распространиться на операторы стоящие выше, что приведёт к ошибке в оценке стоимости исполнения всего плана.

Распределение данных по столбцам – оказывает влияние на оценку селективности соединения, размера выходного отношения. Зная распределение, оптимизатор может делать предположения, как данных хранятся в отношениях: использовать последовательное сканирование или индекс для поиска(и нужно ли строить индекс).

Методы ускорения перебора соединений таблиц

СУБД применяют классические подходы к оптимизации порядка соединений. Классические алгоритмы включают в себя:

- Методы ДП, выполняющие полный поиск среди всех возможных порядков соединения, но требуют больших затрат памяти и зачастую имеют плохую асимптотическую сложность.
- Эвристические методы, требуют меньших затрат ресурсов по сравнению с ДП, но находят приблизительно (локально) оптимальные планы, а могут также получать и неоптимальные.

Классические подходы

Основная идея методов ДП в том, чтобы разбить задачу поиска оптимального порядка на меньшие(по размеру соединяемых отношений) и решить оптимально их. Затем объединить в более крупные и так далее, пока не получим оптимальное решение для текущего запроса.
Эвристический поиск по определённому предположению выбирает "выгодные" планы, отсеивая "плохие" на каждом шаге. Зачастую это позволяет получить приблизительно оптимальный план.
В данной работе рассмотрим следующие алгоритмы ДП: DPsize, DPsub, DPccp и DPhyp. А также эвристические алгоритмы GOO(Greedy Operator Ordering) и Geqo. Помимо этого разберём как работает планировщик в **PostgreSQL**.

DPsize [Neu, стр. 157]

Строит оптимальное **ветвистое дерево** (такое дерево, что хотя бы у одной вершины, начиная с корня, оба потомка составные отношения, т.е не являются таблицами) снизу вверх, начиная с маленьких соединений и расширяя их. Имеет существенное ограничение – не поддерживает работу с внешними соединениями, т.к их не всегда можно переупорядочивать.
Изначально В хранит каждое отношение R_i как лучший план для R_i. Затем начинается перебор всех подмножеств размера $|s|$ возможных планов от 2 до n. Перебираются всевозможные разбиения S на непересекающиеся множества S_1, S_2, такие что существует хотя бы пара отношений в S_1 и S_2, связанная между собой условием соединения. Так как $|S_1|$ и $|S_2|$ меньше $|s|$, то известны их оптимальные планы p_1 и p_2 соответственно.

Стоимость объединения p_1 и p_2 меньше дешевле старой комбинации S_1 и S_2, то происходит замена.
Сложность DPsize [Moe]:

$$I^{\text{chain}}_{\text{DPsize}}(n) = \begin{cases} \frac{1}{48}(5n^4 + 6n^3 - 14n^2 - 12n), & n \text{ even} \\ \frac{1}{48}(5n^4 + 6n^3 - 14n^2 - 6n + 11), & n \text{ odd} \end{cases}$$

$$I^{\text{cycle}}_{\text{DPsize}}(n) = \begin{cases} \frac{1}{4}(n^4 - n^3 - n^2), & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n), & n \text{ odd} \end{cases}$$

$$I^{\text{star}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-4} - \frac{1}{4}\binom{2n}{n-1} + q(n), & n \text{ even} \\ 2^{2n-4} - \frac{1}{4}\binom{2(n-1)}{(n-1)} + \frac{1}{4}\binom{(n-1)}{(n-1)/2} + q(n), & n \text{ odd} \end{cases}$$

$$\text{with } q(n) = n2^{2n-1} - 5 \times 2^{n-3} + \frac{1}{2}(2^n - 5n + 4)$$

$$I^{\text{clique}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-2} - 5 \times 2^{n-2} + \frac{1}{4}\binom{2n}{n} - \frac{1}{4}\binom{n}{n/2} + 1, & n \text{ even} \\ 2^{2n-2} - 5 \times 2^{n-2} + \frac{1}{4}\binom{2n}{n} + 1, & n \text{ odd} \end{cases}$$

Input: A set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: An optimal bushy join tree
$B \leftarrow$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$ **do**
    $B[\{R_i\}] \leftarrow R_i$
end for
for each $1 < s \leq n$ **ascending do**
    for each $S_1, S_2 \subset R$ **such that** $|S_1| + |S_2| = s$ **do**
        if (**not** cross products $\wedge \neg S_1$ connected to S_2) $\vee$ $(S_1 \cap S_2 \neq \emptyset)$ **then**
            continue
        end if
        $p_1 \leftarrow B[S_1], p_2 \leftarrow B[S_2]$
        if $p_1 = \epsilon$ **or** $p_2 = \epsilon$ **then continue**
        end if
        $P \leftarrow$ CreateJoinTree(p_1, p_2)
        if $B[S_1 \cup S_2] = \epsilon$ **or** $C(B[S_1 \cup S_2]) > C(P)$ **then**
            $B[S_1 \cup S_2] \leftarrow P$
        end if
    end for
end for

Где **chain** - запросы соединения от n таблиц, имеющие вид цепочки. **Cycle, star, clique** – цикл, звезда(одна вершина связана со многими, все остальные между собой не имеют связей), полный граф.

DPsub [Neu, стр. 158]

```
1: Input: A set of relations R = {R_1, ..., R_n} to be joined
2: Output: An optimal bushy join tree
3: B ← an empty DP table 2^R → join tree
4: for each R_i ∈ R do
5:     B[{R_i}] ← R_i
6: end for
7: for each 1 < i ≤ 2^n − 1 ascending do
8:     S ← {R_j ∈ R | (⌊i/2^{j−1}⌋ mod 2) = 1}
9:     for each S_1, S_2 ⊂ S such that S_2 = S \ S_1 do
10:        if (not cross products ∧¬S_1 connected to S_2) then
11:            continue
12:        end if
13:        p_1 ← B[S_1], p_2 ← B[S_2]
14:        if p_1 = ϵ or p_2 = ϵ then continue
15:        end if
16:        P ← CreateJoinTree(p_1, p_2)
17:        if B[S] = ϵ or C(B[S]) > C(P) then
18:            B[S] ← P
19:        end if
20:    end for
21: end for
```

Перебирает всевозможные подмножества таблиц S. Разделение множество S на S_1 и S_2, так же как в DPsize. Аналогично, имеются лучшие порядки соединений p_1 и p_2, и если дерево, построенное из p_1 и p_2 дешевле старого варианта то происходит замена. Сложность:

$$I^{\text{chain}}_{\text{DPsub}}(n) = 2^{n+2} - n^2 - 3n - 4$$
$$I^{\text{cycle}}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2$$
$$I^{\text{star}}_{\text{DPsub}}(n) = 2 \times 3^{n-1} - 2^n$$
$$I^{\text{clique}}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1$$

DPccp [Neu, стр. 168] [Moe]

Пусть дан граф соединений G = (V, E). Введём понятие csg-cmp-pair (S1, S2) – в графе запроса выделим S_1, S_2 – связные непересекающиеся подграфы, такие что $S_1 \in V$, $S_2 \in V/S_1$ (отсутствие пересечения) и существует между ними ребро. S_1, S_2 в (S_1, S_2) называются комплементарной парой.

Определим **csg** – количество связных подграфов, в определении csg-cmp-pair это S_1 или S_2.

Определим **ccp** – количество csg-cmp-pairs.

Требуется, чтобы алгоритм обрабатывал все соединения: если оператор коммутативен, то для построения плана для $S_1 \cup S_2$ из планов для S_1 и S_2 нужно учитывать коммутативность.

Однако это не представляет особой сложности. Для того, чтобы быть корректным, любой алгоритм ДП должен учитывать все csg-cmp-pairs. Таким образом, минимальное количество вызовов функции стоимости любого алгоритма ДП в точности равно количеству csg-cmp-pairs для данного графа. В дальнейшем, в алгоритмах использующих понятие csg-cmp-pair, csg будет количеством планов хранимых в DP таблице, как решения меньших подзадач. Ccp - будет нижней возможной оценкой алгоритма, так как в процессе работы ему нужно обработать все csg-cmp-pairs.

Заметим, что если (S_1, S_2) - csg-cmp-pair, то и (S_2, S_1) – тоже. Мы ограничим перечисление csg-cmp-pairs теми (S_1, S_2), которые удовлетворяют условию, что $\min(S_1) < \min(S_2)$, где $\min(S) = s$ такое, что $s \in S$ и $\forall s^{'} \in S : s \neq s^{'} \implies s < s^{'}$. Поскольку это ограничение будет действовать для всех csg-cmp-pairs, обработанных алгоритмом, не будет вычислено ни одной дублирующей csg-cmp-pair.
Теперь задача состоит в том, чтобы перечислить csg-cmp-pairs эффективно и в порядке, приемлемом для ДП. Задача может быть выражена более конкретно. Перед перечислением csg-cmp-pair (S_1, S_2), все csg-cmp-pairs $(S_1^{'}, S_2^{'})$, где $S_1^{'} \subseteq S_1$ и $S_2^{'} \subseteq S_2$ должны быть пронумерованы.
Сначала алгоритм инициализирует начальные значения $B[R_i] = R_i$. Затем перебираются такие подмножества вершин S_1 и S_2, что (S_1, S_2) – csg-cmp-pair и $B[S_1]$, $B[S_2]$ уже известны. Берутся оптимальные планы p_1 и p_2 от S_1 и S_2, затем они соединяются. Если стоимость получившегося плана больше B[S], то происходит замена. Ограничения:

- Не поддерживаются внешние соединения, из-за разбиения на csg-cmp-pairs.
- Сильная зависимость алгоритма от эффективности поиска csg-cmp-pair.

1: **Input:** A set of relations $R = \{R_1, \ldots, R_n\}$
2: **Output:** An optimal bushy join tree
3: $B \leftarrow$ an empty DP table $2^R \rightarrow$ join tree
4: **for each** $R_i \in R$ **do**
5: $\quad B[\{R_i\}] \leftarrow R_i$
6: **end for**
7: **for each** csg-cmp-pairs (S_1, S_2), S = $S_1 \cup S_2$ **do**
8: $\quad p_1 \leftarrow B[S_1], p_2 \leftarrow B[S_2]$
9: $\quad P \leftarrow$ CreateJoinTree(p_1, p_2)
10: $\quad$ **if** $B[S] = \epsilon$ **or** $C(B[S]) > C(P)$ **then**
11: $\quad\quad B[S] \leftarrow P$
12: $\quad$ **end if**
13: **end for**
14: **return** $B[\{R_0, \ldots, R_{n-1}\}]$

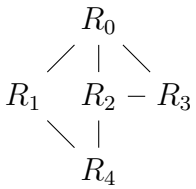
Данная реализация возможна, только если удастся перечислить все csg-cmp-pairs, т.к DPccp перебирает только связные графы и их комплементарные пары. Нужно выполнить несколько задач:

- Эффективно пронумеровать все csg.
- Не пересчитывать csg дважды, пример (S1, S2) и (S2, S1).
- Эффективно находить ccp для каждой csg.

Предварительно проведём поиск в ширину и пронумеруем все вершины: $V = R_0, \ldots, R_{n-1}$. Пусть G = V, E – граф запроса. Введём множества $B_i = \{v_j | j <= i\}$ и функцию соседства $N(V') = \{v' | v \in V' (v, v') \in E\}$, чтобы избавиться от повторного добавления (S_1, S_2) и (S_2, S_1)

Algorithm EnumerateCsg

1: **for** $i = n - 1$ **to** 0 **do**
2: **emit** u_i; – Добавим вершины
3: EnumerateCsgRec(G, u_i, B_i);
4: **end for**
5: EnumerateCsgRec(G, S, X);
6: $N \leftarrow \mathcal{N}(S) \setminus X$;
7: **for each** $S' \subseteq N$ **where** $S' \neq \emptyset$ **do**
8: **emit** $(S \cup S')$ – Добавим подграф
9: **end for**
10: **for each** $S' \subseteq N$ **where** $S' \neq \emptyset$ **do**
11: EnumerateCsgRec$(G, (S \cup S'), (X \cup N))$
12: **end for**



S	X	N	emit/S
$\{4\}$	$\{0,1,2,3,4\}$	$\emptyset$	
$\{3\}$	$\{0,1,2,3\}$	$\{4\}$	$\{3,4\}$
$\{2\}$	$\{0,1,2\}$	$\{3,4\}$	$\{2,3\}$ $\{2,4\}$ $\{2,3,4\}$
$\{1\}$	$\{0,1\}$	$\{4\}$	$\{1,4\}$
$\rightarrow \{1,4\}$	$\{0,1,4\}$	$\{2,3\}$	$\{1,2,4\}$ $\{1,3,4\}$ $\{1,2,3,4\}$
$\{0\}$	$\{0\}$	$\{1,2,3\}$	$\{0,1\}$ $\{0,2\}$ $\{0,3\}$
$\rightarrow \{0,1\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,1,4\}$
$\rightarrow \{0,2\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,2,4\}$
$\rightarrow \{0,3\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,3,4\}$

Вместе эти алгоритмы позволяют эффективно находить ccp для каждой csg. Сложность:

$$I^{\text{chain}}_{\text{DPchain}}(n) = n^3$$

Algorithm EnumerateCmp

1: **Input:** A connected query graph $G = (V, E)$, a connected subset S_1
2: **Precondition:** Nodes in V are numbered according to a BFS
3: **Output:** Emits all complements S_2 for S_1 such that (S_1, S_2) is a csg-cmp-pair
4: $X \leftarrow B_{\min(S_1)} \cup S_1$
5: $N \leftarrow \mathcal{N}(S) \setminus X$
6: **for each** $u_i \in N$ **by descending** i **do**
7: **emit** u_i
8: EnumerateCsgRec$(G, \{u_i\}, X \cup N)$
9: **end for**

$$I^{\text{cycle}}_{\text{DPccp}}(n) = n^3$$

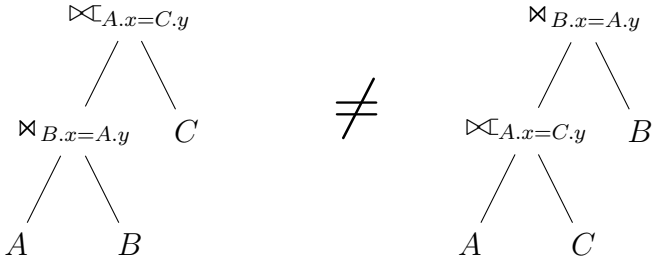
$$I^{\text{star}}_{\text{DPccp}}(n) = n2^n$$

$$I^{\text{clique}}_{\text{DPccp}}(n) = 3^n$$

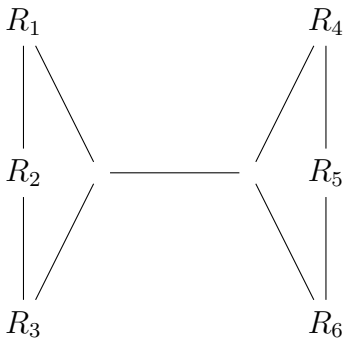
Можно заметить, что также как DPsize и DPsub, DPccp не поддерживает внешние соединения. Однако он на каждом типе графа имеет меньшую сложность и имеет более сложную реализацию чем DPsize и DPsub.

DPhyp [GM08]

Является улучшенной версией алгоритма DPccp для работы с внешними соединениями, которые не всегда коммутативны, а значит не всегда допускают переупорядочивания – есть ограничение на порядок исполнения. Это значит, что если мы сделаем невалидное переупорядочивание операторов соединения, результат изменится.



При работе с внешними соединениями возникает понятие гиперграфа.
Гиперграф – пара (V, E), где V – непустое множество вершин и E – множество гиперрёбер. **Гиперребро** – неупорядоченная пара (u, v) непустых подмножеств V($u \subset V, v \subset V$), с условием $u \cap v = \emptyset$ Пусть все вершины V линейно упорядочены по отношению $<$.



Пример: $R_1.a + R_2.b + R3.c = R_4.d + R_5.e + R_6.f$. Этому условию соединения соответствует гиперребро $(\{R_1, R_2, R_3\}, \{R_4, R_5, R_6\})$. $V = R_1, \ldots, R_6$. Простые рёбра: $(R_1, R_2), (R_2, R_3), (R_4, R_5)$ и (R_5, R_6). Отшошение линейного порядка : $R_i < R_j \Leftrightarrow i < j$.
Подграф – есть гиперграф H(V, E), $V^{'} \subseteq V$ и $E^{'} = \{(u, v) | (u, v) \in E, u \in V^{'} v \in V^{'}\}$. Тогда гиперграф $G^{'} = (V^{'}, E^{'})$, порождённый $V^{'}, E^{'}$ будет подграфом.

Связность гиперграфа — пусть H — гиперграф. H связный, если $|V| = 1$ или существует разбиение V на V' и V'' и существует гиперребро $(u, v) \in E$: $u \subseteq V'$ и $v \subseteq V''$, и порождённые V' и V'' подграфы связные. Краткое обозначение: **csg**.
Связное дополнение — $V' \subseteq V$, $V'' \subseteq (V \setminus V')$ и порождённые V' и V'' подграфы связные, то V'' — связное дополнение V'. Краткое обозначение: **cmp**.
Csg-cmp-pair для гиперграфа — пусть $H = (V, E)$ — гиперграф, а S_1, S_2 — два подмножества V такие, что $S_1 \subseteq V$ и $S_2 \subseteq (V \setminus S_1)$ являются связным подграфом и связным дополнением. Если существует гиперребро $(u, v) \in E$ такое, что $u \subseteq S_1$ и $v \subseteq S_2$, то мы называем (S_1, S_2) **csg-cmp-pair**.
Основная идея **Dphyp** такая же, как и у **DPccp**: постепенно расширять **csg**-графы, беря новые вершины из функции соседства N. $N(S, X)$ при исключающем множестве X состоит из всех вершин, достижимых из S, которые не входят в X. При этом, выбирая подмножества соседства для включения, мы должны рассматривать гиперузел как атомарный экземпляр: либо все его вершины находятся внутри перечисляемого подмножества, либо ни один из них.
Так как гиперузлы могут пересекаться и упорядоченность вершин важна, определим минимальный элемент множества:

$$\min(S) = \{s \mid s \in S, \forall s' \in S : s \neq s' \Rightarrow s \prec s'\}.$$

Для определения функции соседства $N(S, X)$ для гиперграфа определим минимальное множество $E^{\downarrow}$, такое, что для любого гиперребра $(u, v) \in E$ существует гиперребро $(u', v') \in E^{\downarrow}$, где $u' \subseteq u$ и $v' \subseteq v$.
Но для начала определим множество:

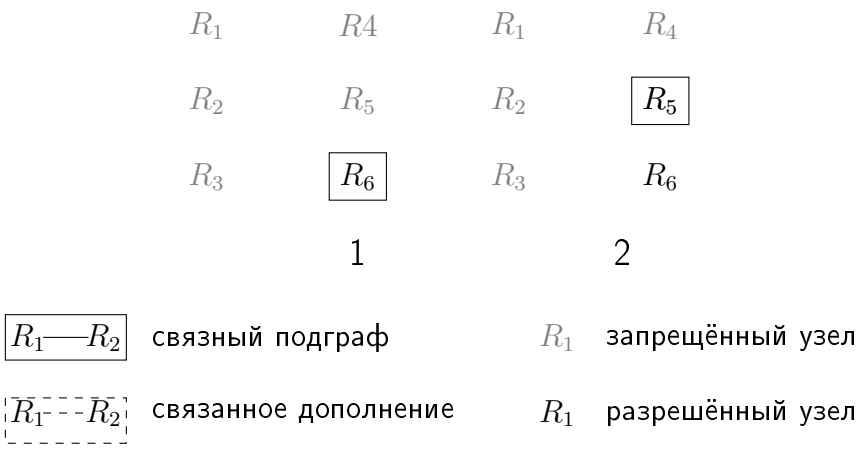
$$E^{\downarrow'}(S, X) = \{v \mid (u, v) \in E, u \subseteq S, v \cap S = \emptyset, v \cap X = \emptyset\}.$$

Определим $E^{\downarrow}(S, X)$ как минимальное множество гиперузлов, такое, что для всех $v \in E^{\downarrow'}(S, X)$ существует гиперузел $v' \in E^{\downarrow}(S, X)$, такой что $v' \subseteq v$.
Теперь для гиперграфа определим функцию соседства: N(S, X) = $\bigcup_{v \in E^{\downarrow}(S,X)} \min(v)$
Перейдём к самому алгоритму DPhyp, основная идея заключается:

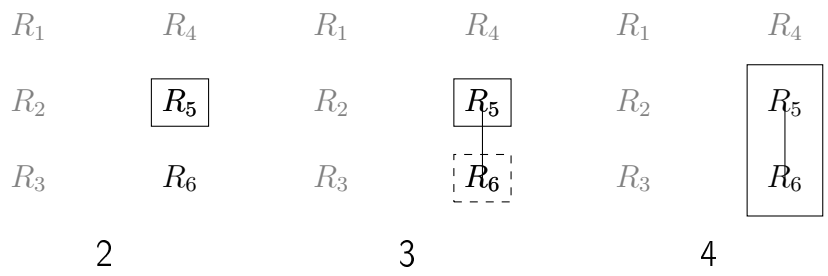
- Алгоритм строит все **ccp** путём перечисления связных подграфов, добавляя большие(по отношению <) вершины к подграфам.
- Csg и связанные с ними cmp создаются путём рекурсивного обхода гиперграфа.
- Во время обходя графа, некоторые вершины запрещены для посещения, чтобы избежать дублирования csg-cmp-pairs.
- Csg увеличиваются за счёт ребёр, ведущим к соседним вершинам. Конечная сторона ребра может приводить сразу к нескольким примитивным вершинам, что нарушает рекурсивный рост компонентов. Поэтому гиперрёбра интерпретируются как n к 1 рёбра: n рёбер с одной стороны ведут к одной вершине с другой стороны. Затем с этой вершины на конце ребра происходит рекурсивный рост и используется таблица DP, чтобы проверить, достигнуто ли допустимое построение.

Алгоритм начинается с Solve, которая инициализирует DP таблицу с планами доступа для одиночных отношений, а затем вызывает EmitCsg и EnumerateCsgRec для каждого набора, содержащего ровно одно отношение. EnumerateCsgRec отвечает за перечисление csg. Для этого она вычисляет соседство и перебирает каждое ее подмножество. Для каждого такого подмножества S_1 вызывается EmitCsg – нужна для нахождения подходящего дополнения. Для этого нужно вызвать EnumerateCmpRec, который рекурсивно перечисляет дополнения S_2 для найденного ранее csg S_1. Пара (S_1, S_2) является csg-cmp-pair. Для каждой такой пары вызывается EmitCsgCmp. Ее основная задача – рассмотреть план, построенный из планов для S_1 и S_2.

```
1: Solve()
2: for each u ∈ V do
3:     dpTabele[u] = plan for u
4: end for
5: for each u ∈ V descending according to < do
6:     EmitCsg(u) - process singleton sets
7:     EnumerateCsgRec(u,B_u) - expand singleton sets
8: end for
9: return dpTabele[V]
```



В первом цикле инициализируется таблица ДП с планами для одиночных отношений. Во втором цикле для каждой вершины графа запросов в порядке убывания (согласно $\prec$) вызываются `EmitCsg` и `EnumerateCsgRec`.
Алгоритм вызывает `EmitCsg`$(\{v\})$ для отдельных вершин $v \in V$, чтобы построить все **csg-cmp-pairs** $(\{v\}, S_2)$ через вызовы `EnumerateCsgCmp` и `EmitCsgCmp`, где $v \prec \min(S_2)$. Это условие подразумевает, что каждая **csg-cmp-pair** создаётся только один раз, и симметричные пары не генерируются.
Это соответствует одновершинным графам, например, шаг 1 и 2.
Вызовы `EnumerateCsgRec` расширяют исходное множество $\{v\}$ в большие множества S_1, для которых затем находятся связные подмножества его дополнения S_2, такие, что (S_1, S_2) приводит к **csg-cmp-pair**.
На шаге 2, где `EnumerateCsgRec` начинается с R_5 и расширяет его до $\{R_5, R_6\}$ на шаге 4 (шаг 3 — это построение дополнения).

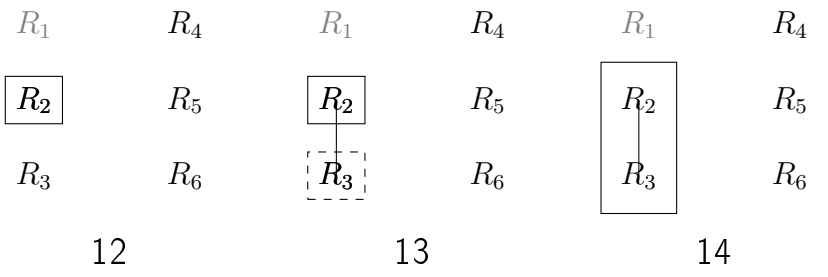


Чтобы избежать дублирования при перечислении, все вершины, которые упорядочены перед v согласно $<$, запрещены при рекурсивном расширении. Формально мы определяем это множество как $B_v = \{w|w < v\} \cup v$.

EnumerateCsgRec(S_1, X)

1: **for** each $N \subseteq \mathcal{N}(S_1, X) : N \neq \emptyset$ **do**
2: **if** $dpTable[S_1 \cup N]$ **then**
3: EmitCsg($S_1 \cup N$)
4: **end if**
5: **end for**
6: **for** each $N \subseteq \mathcal{N}(S_1, X) : N \neq \emptyset$ **do**
7: EnumerateCsgRec($S_1 \cup N, X \cup \mathcal{N}(S_1, X)$)
8: **end for**

Цель `EnumerateCsgRec` — расширить заданное множество S_1, которое порождает связный подграф G, до большего множества с тем же свойством. Для этого рассматривается каждое непустое подходящее подмножество соседств S_1.
Для каждого из этих подмножеств N проверяется, является ли $S_1 \cup N$ связной компонентой. Это делается с помощью поиска в таблице `dpTable`. Если проверка прошла успешно, то найден новый связный компонент, который далее обрабатывается вызовом `EmitCsg`$(S_1 \cup N)$.
Затем, на втором этапе, для всех этих подмножеств N соседства вызывается `EnumerateCsgRec` так, что $S_1 \cup N$ может быть рекурсивно расширен.
Причина, по которой сначала вызывается `EmitCsg`, а затем `EnumerateCsgRec`, заключается в том, что для того, чтобы иметь последовательность перечислений, допустимую для DP, сначала должны быть построены меньшие наборы.

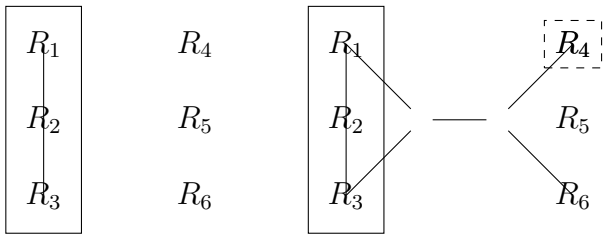


На шаге 12 был построен `Solve` на $S_1 = \{R_2\}$. Соседство состоит только из $\{R_3\}$, так как R_1 находится в X (R_4, R_5, R_6 тоже не в X, так как не достижимы).
`EnumerateCsgRec` сначала вызывает `EmitCsg`, который создаст присоединяемое дополнение (шаг 13). Затем он проверяет $\{R_2, R_3\}$ на связность.
Соответствующая запись в `dpTable` была получена на шаге 13. Следовательно, этот тест проходит успешно, и $\{R_2, R_3\}$ далее обрабатывается рекурсивным вызовом `EnumerateCsgRec` (шаг 14).

EmitCsg(S_1)

```
1: X = S_1 ∪ B_{min(S_1)}
2: N = 𝒩(S_1, X)
3: for each u ∈ N descending according to < do
4:     S_2 = u
5:     if ∃(u, v) ∈ E : u ⊆ S_1 ∩ v ⊆ S_2 then
6:         EmitCsgCmp(S_1, S_2)
7:     end if
8:     EnumerateCmpRec(S_1, S_2, X)
9: end for
```

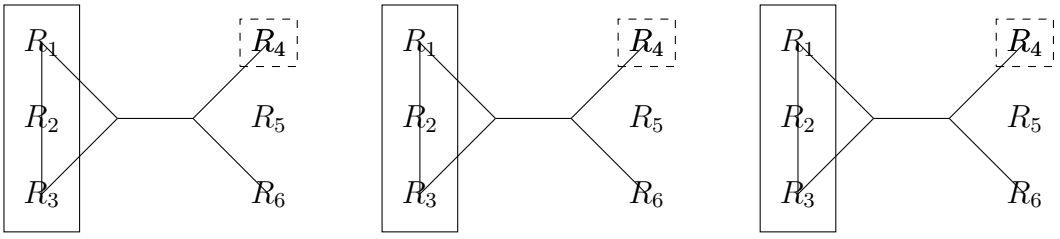
Теперь расширение останавливается, так как соседство $\{R_2, R_3\}$ пусто, поскольку $R_1 \in X$.
`EmitCsg` принимает в качестве аргумента непустое подходящее подмножество S_1 из V, которое порождает связный подграф. Его задача — построить все вершины S_2, такие, что (S_1, S_2) становится **csg-cmp-pair**.
Вершины берутся из соседства S_1. Все вершины, которые упорядочивались до наименьшего элемента в S_1 (охваченного множеством $B_{\min}(S_1)$), удаляются из соседства, чтобы избежать дублирования перечислений.
Поскольку соседство также содержит $\min(v)$ для гиперрёбер (u, v) с $|v| > 1$, то не гарантируется, что S_1 соединён с v. Чтобы избежать построения ложных **csg-cmp-pairs**, `EmitCsg` проверяет их на связность.
Тем не менее соседство из одной вершины может быть расширено до допустимого дополнения S_2 к S_1. Поэтому перед вызовом `EnumerateCmpRec`, который выполняет это расширение, такая проверка не требуется.



Шаг 20. Текущий набор $S_1 = \{R_1, R_2, R_3\}$, а $N = \{R_4\}$. Поскольку нет гиперребра, соединяющего эти два множества, вызов `EmitCsgCmp` не требуется.
Однако множество $\{R_4\}$ может быть расширено до допустимого дополнения, а именно $\{R_4, R_5, R_6\}$. Подходящие вершины для дополнения является задачей вызова `EnumerateCmpRec` на шаге 21.
`EnumerateCsgRec` имеет три параметра. Первый параметр S_1 используется только для передачи его в `EmitCsgCmp`.
Второй параметр — это множество S_2, которое является связным и должно расширяться до тех пор, пока не будет достигнута допустимая **csg-cmp-pair**. Поэтому рассматривается соседство S_2.
Для каждого непустого подходящего подмножества N соседства проверяется, является ли $S_2 \cup N$ связным подмножеством и соединённым с S_1.
Если да, то мы имеем корректную **csg-cmp-pair** (S_1, S_2) и можем приступить к построению плана (это делается в `EmitCsgCmp`).
Независимо от результата проверки, мы рекурсивно пытаемся расширить S_2, так, чтобы эта проверка оказалась успешной.
В целом, `EnumerateCmpRec` очень похоже на `EnumerateCsgRec`.

EnumerateCmpRec(S_1, S_2, X)

1: **for all** $N \subseteq \mathcal{N}(S_2, X)$: $N \neq \emptyset$ **do**
2: **if** dpTable$[S_2 \cup N] \neq \emptyset \wedge \exists (u, v) \in E$: $u \subseteq S_1 \ \wedge \ v \subseteq S_2 \cup N$ **then**
3: EmitCsgCmp$(S_1, S_2 \cup N)$
4: **end if**
5: **end for**
6: $X \leftarrow X \cup \mathcal{N}(S_2, X)$
7: **for all** $N \subseteq \mathcal{N}(S_2, X)$: $N \neq \emptyset$ **do**
8: EnumerateCmpRec$(S_1, S_2 \cup N, X)$
9: **end for**



На шаге 21 $S_1 = \{R_1, R_2, R_3\}$ и $S_2 = \{R_4\}$. $N = \{R_5\}$.
Множество $\{R_4, R_5\}$ порождает связный подграф. Он был вставлен в dpTable на одном из прошлых шагов. Однако не существует гиперребра, соединяющего его с S_1.
Следовательно, вызов EmitCsgCmp не производится.
Далее следует рекурсивный вызов на шаге 22 с изменением S_2 на $\{R_4, R_5\}$. Его соседством является $\{R_6\}$. Множество $\{R_4, R_5, R_6\}$ приводит к связному подграфу.
Поиск в dpTable находит соответствие, так как запись была сделана на шаге 7. Вторая часть проверки также успешна, так как одно гиперребро соединяет это множество с S_1.
Следовательно, вызов EmitCsgCmp на шаге 23 происходит и строит планы, содержащие все отношения.

EmitCsgCmp(S_1, S_2)

1: plan$_1$ = dpTable$[S_1]$
2: plan$_2$ = dpTable$[S_2]$
3: $S = S_1 \cup S_2$
4: $p = \bigwedge_{(u_1,u_2)\in E, u_i \subseteq S_i} \mathcal{P}(u_1, u_2)$
5: newplan = plan$_1 \bowtie_p$ plan$_2$
6: **if** dpTable$[S] = \emptyset \ \vee$ cost(newplan) $<$ cost(dpTable$[S]$) **then**
7: dpTable$[S]$ = newplan
8: **end if**
9: newplan = plan$_2 \bowtie_p$ plan$_1$ // for commutative ops only
10: **if** cost(newplan) $<$ dpTable$[S]$ **then**
11: dpTable$[S]$ = newplan
12: **end if**

Задача EmitCsgCmp(S_1, S_2) — соединить оптимальные планы для S_1 и S_2, которые должны образовать **csg-cmp-pair**.
Для этого мы должны быть в состоянии вычислить правильный предикат соединения и затраты на результирующие соединения. Для этого необходимо, чтобы предикаты присоединения, селективности и кардинальности были привязаны к гиперграфу.

Поскольку мы скрываем вычисления стоимости в абстрактной функции `cost`, нам нужно только явно собрать предикат соединения.
Для данного гиперграфа $G = (V, E)$ и гиперребра $(u, v) \in E$, определим $\mathcal{P}(u, v)$ — условие соединения по этому ребру.
Сначала из таблицы DP извлекаются оптимальные планы для S_1 и S_2. Затем мы запоминаем в S общее множество отношений, присутствующих в плане, который необходимо построить.
Предикат соединения p собирается путём взятия конъюнкции предикатов тех гипергранespecially, которые соединяют S_1 и S_2.
Затем планы строятся, и если они дешевле существующих, они сохраняются в `dpTable`.

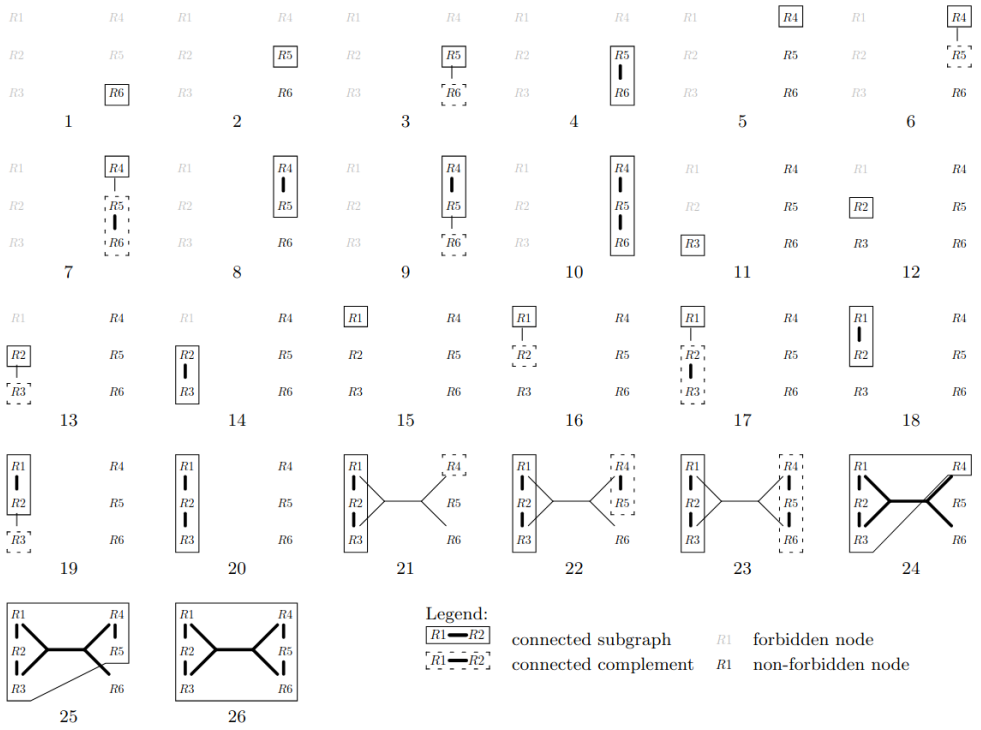
Обработка внешних соединений

Введём два понятия из статьи:

Syntactic eligibility set (SES) – множество таблиц, которые должны быть соединены до того, как можно применить внешнее соединение.

Total eligibility set (TES) – полный список ограничений на порядок , расширяет SES и учитывает все зависимости между соединениями. Формирует гиперрёбра, запрещая недопустимые перестановки. Строится на основе SES и обходе логического дерева соединений, проверяя конфликты с другими операторами. Добавляя TES в гиперграф можно создавать допустимые рёбра.

Использование гиперграфа позволяет запрещать недопустимые перестановки внешних соединений, задавая при этом допустимый порядок соединений т.е ненужные рёбра не будут сгенерированы в графе.
Таким образом DPhyp, является относительно(DPsize, DPsub) эффективным алгоритмом. При этом позволяет обрабатывать внешние соединения.
Все рассмотренные алгоритмы порождали планы в виде ветвистых деревьев, так как они в отличие от односторонних деревьев позволяют использовать преимущества параллелизма. Планы можно выполнить параллельно, если они не зависят друг от друга, т.е соединения A и B можно выполнить параллельно, если на любом пути от корня до таблицы нет одновременно A и B. Такого нельзя добиться на односторонних деревьях.



ДП в PostgreSQL

В PostgreSQL в файле src/backend/optimizer/path/allpaths.c реализованы алгоритмы планирования запросов. Они применяются в методе:

```
static RelOptInfo *make_rel_from_joinlist(PlannerInfo  *root,
   List  *joinlist){...}
```

Внутри метода есть переменная int $levels\_needed$ – количество соединяемых таблиц. Также есть блок кода в котором происходит выбор подхода для оптимизации:

```
    if (join_search_hook)
        return (*join_search_hook) (root, levels_needed,
   initial_rels);
    else if (enable_geqo && levels_needed >= geqo_threshold)
        return geqo(root, levels_needed, initial_rels);
    else
        return standard_join_search(root, levels_needed,
   initial_rels);
```

Нас интересует метод ДП $standard\_join\_search$, geqo опишем позже. Основная идея этого метода заключается в следующих шагах:

1. Сначала обрабатываются отдельные таблицы (базовые RelOptInfo).
2. Строится все возможные соединения по два (2-way joins).
3. Комбинирование с уже построенными соединениями в большие группы (3-way, 4-way и т. д.).
4. После построени k-way отбрасываются неэффективные соединения.
5. В конце находится самый дешёвы путь.

Для хранения всех путей из k отношений имеется $root->join\_rel\_level[k]$.

В начале происходит инициализация списком начальных отношений из запроса:
$root->join\_rel\_level[1] = initial\_rels$
Строим всевозможные соединения шаг за шагом (итерации от lev = 2 до $levels\_needed$), основываясь на меньших по размеру результатах:

```
1     for (lev = 2; lev <= levels_needed; lev++) ...
```

Внутри цикла в начале вызывается метод $join\_search\_one\_level(root, lev)$, который строит всевозможные соединения для текущего уровня lev, основываясь на результатах прошлых вызовов для меньших наборов отношений.
Для этого:

1. Создаются соединения между level-1 – табличными группами и одиночными отношениями(таблицами) — строятся левосторонние, правосторонние пути, также учитываются внешние соединения.

2. Если не нашлось соединений по условиям, то производится декартовое произведение отношений.

3. Рассматриваются ветвистые планы – соединения двух групп в цикле – размеров k и N-k, также учитываются внешние соединения.

4. Если для текущего уровня ничего не получилось построить – делается декартовое произведение отношений.

Построение односторонних планов.

```
1     foreach(r, joinrels[level - 1])
2   {
3     RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
4     if (old_rel->joininfo != NIL || old_rel->has_eclass_joins
    ||
5       has_join_restriction(root, old_rel))
6     {
7       int first_rel;
8       if (level == 2) /* consider remaining initial rels */
9         first_rel = foreach_current_index(r) + 1;
10      else
11        first_rel = 0;
12      make_rels_by_clause_joins(root, old_rel, joinrels[1],
    first_rel);
13    }
14    else
15    {
16      make_rels_by_clauseless_joins(root,old_rel, joinrels[1]);
17    }
18  }
```

Построение ветвистых планов.

```
1     for (k = 2;; k++)
2   {
```

```
    int other_level = level - k;
    if (k > other_level)
      break;
    foreach(r, joinrels[k])
    {
      RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);
      int first_rel;
      ListCell *r2;
      if (old_rel->joininfo == NIL && !old_rel->
  has_eclass_joins &&
        !has_join_restriction(root, old_rel))
        continue;
      if (k == other_level) /* only consider remaining rels */
        first_rel = foreach_current_index(r) + 1;
      else
        first_rel = 0;
      for_each_from(r2, joinrels[other_level], first_rel)
      {
        RelOptInfo *new_rel = (RelOptInfo *) lfirst(r2);
        if (!bms_overlap(old_rel->relids, new_rel->relids))
        {
          if (have_relevant_joinclause(root, old_rel, new_rel)
  ||
            have_join_order_restriction(root, old_rel, new_rel)
  )
          {
            (void) make_join_rel(root, old_rel, new_rel);
          }
        }
      }
    }
  }
```

Так как $make\_join\_rel(x, y)$ создаёт оба варианта соединений ((x, y) и (y, x)), берём только половину возможных комбинаций, чтобы не делать дублирующую работу.
Если ничего не смогли построить на этом уровне, то делаем декартовое произведение.

```
    if (joinrels[level] == NIL)
  {
    foreach(r, joinrels[level - 1])
    {
      RelOptInfo *old_rel = (RelOptInfo *) lfirst(r);

      make_rels_by_clauseless_joins(root,
                    old_rel,
                    joinrels[1]);
    }
    if (joinrels[level] == NIL &&
      root->join_info_list == NIL &&
      !root->hasLateralRTEs)
      elog(ERROR, failed to build any %d-way joins, level);
```

```
15    }
```

После завершения работы $join\_search\_one\_level$, нужно провести оптимизацию и получить оптимальные планы для текущего уровня. Для этого используется:

- Оптимизация $partitionwise\_join$.
- Использование параллелизма.
- Выбрать самый дешёвый путь.

Для этого проводится обход всех созданных соединений на уровне lev.

```
1     foreach(lc, root->join_rel_level[lev])
2     {
3         rel = (RelOptInfo *) lfirst(lc);
4         /* Create paths for partitionwise joins. */
5         generate_partitionwise_join_paths(root, rel);
6         /*
7          * Except for the topmost scan/join rel, consider
    gathering
8          * partial paths.  We'll do the same for the topmost
    scan/join rel
9          * once we know the final targetlist (see
    grouping_planner's and
10         * its call to apply_scanjoin_target_to_paths).
11         */
12        if (!bms_equal(rel->relids, root->all_query_rels))
13            generate_useful_gather_paths(root, rel, false);
14
15        /* Find and save the cheapest paths for this rel */
16        set_cheapest(rel);
17    }
```

$generate\_partitionwise\_join\_paths$ ищет возможность разбиения соединения по партициям. Она проверяет разбита ли таблица по партициям. Если да, то обходит их и рекурсивно строит пути соединений для каждой партиции, затем добавляет найденные пути.
$generate\_useful\_gather\_paths$ – добавляет пути для параллельного исполнения.
$set\_cheapest$ – выбирает самый дешёвый путь для текущего уровня.
В целом DP алгоритм, реализованный в Postgre, сильно похож на DPsize: в нём также итеративно увеличивается размер множество выбранных таблиц, но в отличие от DPsize, здесь сначала строятся всевозможные пути для текущего размера множества, затем происходит оптимизация. Также в отличие от DPsize PostgreSQL поддерживает обработку внешних соединений и способен проводить дополнительные оптимизации.
Тем не менее, когда количество таблиц для соединения становится большим(например >15) все алгоритмы DP начинают работать непозволительно долго. При этом в современном мире зачастую встречаются запросы, содержащие большое количество соединений таблиц (например фильтры поиска в интернет магазинах или каталогах, запросы сгенерированные машиной). Для решения таких задач прибегают к использованию эвристических алгоритмов.
Рассмотрим теперь эвристические методы планирования запросов. Применяемые когда количество таблиц в запросе велико.

GOO(Greedy Operator Order) [Neu, стр. 108] [All18]

GOO – алгоритм, который выбирает жадно(не рассматривает все варианты порядка соединений, а пытается локально взять лучший порядкой) оптимальное соединение на каждом этапе, формируя итоговый порядок соединений таблиц шаг за шагом. Выходное дерево соединений произвольное, т.е может быть либо ветвистым или односторонним.

- На каждом шаге выбираем два отношения, которых выгоднее всего соединить. За стоимость таблиц i и j можно взять селективность Sel[i,j] cost(i,j) = size(i) * size(j) * Sel[i, j].
- Объединяем их в одно поддерево.
- Повторяем процесс, пока не останется одно дерево соединений.

GOO

1: **Input:** A set of relations $R = \{R_1, \ldots, R_n\}$
2: **Output:** An roughly optimal join tree
3: T = R
4: **while** |T| > 1 **do**
5: select $(T_i, T_j) = arg\min_{T_i,T_j \in T, T_i \neq T_j} cost(T_i, T_j)$
6: $T = T \setminus T_i, T_j$
7: $T = T \cup T_i \bowtie T_j$
8: **end while**
9: return T_0

Данный алгоритм просто в реализации и имеет сложность O(n^3), не поддерживает внешние соединения.

Генетический алгоритм

Это эволюционный метод, который использует принципы естественного отбора и мутаций для поиска оптимального порядка соединений. Имеется множество планов – популяция особей. Каждая особь(план) кодируется хромосомой. Каждая хромосома это пара из последовательности генов и стоимости плана. Ген это оптимизирующий параметр, закодированный в целочисленном представлении. Основные этапы работы алгоритма:

1. **Генерация начальной популяции** – создаётся множество случайных порядков соединений таблиц ("хромосом").
2. **Оценка приспособленности** – измеряется стоимость каждого порядка соединений.
3. **Скрещивание** – комбинируются части выживших порядков соединений, создавая новые последовательности.
4. **Мутация** – небольшие случайные изменения в порядке соединений для поиска лучших решений.
5. **Выбор новых решений** – на каждом этапе остаются только наиболее эффективные планы соединений.
6. **Повторение** 2–4 определённое количество итераций, и выбор самого приспособленного из выживших.

Рассмотрим теперь его реализацию в **PostgreSQL** в *src/backend/optimizer/geqo/geqo_main.c*.
Он вызывается в блоке кода, из того же метода, что и ДП:

```
1     if (join_search_hook)
2         return (*join_search_hook) (root, levels_needed,
    initial_rels);
3     else if (enable_geqo && levels_needed >= geqo_threshold)
4         return geqo(root, levels_needed, initial_rels);
5     else
6         return standard_join_search(root, levels_needed,
    initial_rels);
```

Для его запуска требуется, чтобы была включена опция *enable_geqo* и количество таблиц было больше *geqo_threshold*.
У самого Geqo есть опции:

Gego_effort – Чем больше значение этого параметра, тем больше времени будет потрачено на планирование запроса, но и тем больше вероятность, что будет выбран эффективный план. Является умножающим коэффициентом, от которого зависит размер пула особей, а от него в свою очередь зависит количество итераций.

Gego_generations – Количество итераций. Дефолтное значение 0, если задать больше то переопределяется количество итераций.

Geqo_selection_bias – Влияет на селекцию.

Geqo_seed – начальное значение генератора случайных чисел.

Также в алгоритме есть различные параметры(ERX, PMX, CX, PX, OX1, OX2) – способы скрещивания хромосом. Но только CX допускает мутацию – случайные изменения в хромосоме ребёнка.
Перейдём к самому методу Geqo.

```
1     RelOptInfo * geqo(PlannerInfo *root, int number_of_rels,
    List *initial_rels) {...}
```

Алгоритм начинается с инициализации начального значения для генератора случайных чисел, количества особей *pool_size* и количества итераций алгоритма Затем создаются начальные особи:

```
1     random_init_pool(root, pool);
```

Где pool – массив особей.
Происходит сортировка по возрастанию стоимости:

```
1     sort_pool(root, pool);
```

Запускается цикл по количеству итераций:

1. В материнскую и отцовскую хромосомы записываются рандомно выбранные хромосомы из пула.

```
1         geqo_selection(root, momma, daddy, pool,
    Geqo_selection_bias)
2
```

2. В зависимости от выбранного алгоритма рекомбинации генов, в хромосому ребёнка записывается результат скрещивание родительских хромосом. Если выбран CX, то происходит ещё и мутация.

3. Вычисляется стоимость плана, который кодируется ребёнком.

4. Ребёнок попадает в пул особей(бинарным поиском, не нарушая сортировки), заменяя кого-то.

После всех итераций возвращается первая особь из пула, у неё самая низкая стоимость.

Сравнение методов ускорения перебора соединений

Рекомендации по выбору метода.

После рассмотрения всех алгоритмов, можно сделать вывод, что если в запросе присутствует большое количество соединений таблиц, то для сокращения времени планирования лучше использовать Geqo или Goo. Однако, если количество таблиц невелико (< 15), то лучше использовать DPccp или его модифицированную версию для внешних соединений DPhyp, так как для всех типов графов запросов(Цепочка, цикл, звезда, полный граф) DPccp(DPhyp) имеет лучшую асимптотическую сложность. Но при этом DPsize и DPsub проще в реализации чем DPccp, а тем более и DPhyp

Заключение

Задача поиска оптимального порядка соединений относится к классу NP-полных задач, имеет высокую вычислительную сложность, которая экспоненциально возрастает при увеличении количества соединений в запросе к СУБД. Вместе с этим существует множество факторов влияющих на выбор оптимального порядка. Для решения данной задачи были реализованы используются классические алгоритмы ДП, находящие оптимальный порядок, но требующие значительного количества памяти и сложность, быстро возрастающую при росте числа таблиц, и эвристические методы, находящие приблизительно оптимальный порядок при меньшем потреблении ресурсов.
Данная задача имеет широкие перспективы исследований, так как она имеет значительное число подходов для решения:

- Улучшение классических алгоритмом ДП, позволяющее им работать с внешними соединениями или партициями, создавание планов, допускающих параллельное исполнение, как это сделано в PostgreSQL для DPsize.

- Реализация современного подхода linDP и его модификации linDP++ для обработки внешних соединений. Этот подход позволяет расширить метод ДП на соединение большого числа таблиц: сначала с помощью алгоритма IK/KBZ [TI84] [RK86] построить приблизительно оптимальное левостороннее дерево плана, а затем на его основе построить оптимальное или почти оптимальное ветвистое дерево плана.

- Разработка алгоритмов, которые сочетают ДП и эвристические подходы.

- Разработка и внедрение рандомизированных алгоритмов помимо генетического(является рандомизированным и эвристическим), которые будут производить более оптимальные планы и иметь хорошую сходимость.

- Улучшение существующих эвристических подходов на основе статистик.

- Использование подходов на основе машинного обучения, например Deep Reinforcement Learning, и использование самообучающихся моделей, которые будут использовать историю существующих запросов и статистик для улучшения планов.

Список литературы

[All18] Julian Reddy Allam. Evaluation of a greedy join-order optimization approach using the imdb dataset. *University of Magdeburg*, page 23, 2018.

[GM08] Thomas Neumann Guido Moerkotte. Dynamic programming strikes back. *University of Mannheim, Max-Planck Institute for Informatics*, 2008.

[Liu24] Chang Liu. Query execution plan search space enumeration by reinforcement learning. *School of Electrical Engineering and Computer Science, Faculty of Engineering University of Ottawa*, page 3, 2024.

[Moe] Guido Moerkotte. Dynamic programming for join ordering revisited. *School of Electrical Engineering and Computer Science, Faculty of Engineering University of Ottawa*, page 10.

[Neu] Thomas Neumann. Query optimization. *Technical University of Munich*.

[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.