Методы ускорения перебора соединений таблиц в планировании SQL запросов

НГУ, ММФ, 22126, Пьянзин Богдан

1 марта 2025 г.

Структура курсовой работы

Введение

- Актуальность темы.
- Цель исследования.

Основная часть

Теоретические основы соединений таблиц в SQL запросах

- Определение и классификация соединений и планирование запроса.
- Задача выбора порядка соединений таблиц.
- Факторы, влияющие на производительность (размеры таблиц, статистики, индексы, типы данных и т.д.).

Методы ускорения перебора соединений таблиц

- Классические подходы.
- Реализация в **PostgreSQL**.
- Современные подходы. %TODO

Сравнительный анализ методов ускорения %TODO

- Преимущества и недостатки каждого метода %TODO

Заключение %TODO

- Итоги работы.
- Рекомендации по применению методов.
- Перспективы исследований.

Список литературы

Введение

Актуальность темы

Современные СУБД работают с большим объёмом информации и транзакций, использую 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 зависят от качества сгенерированного физического плана. Для создания "хорошего"плана, нужно решить одну решить или приблизить решение NP полной задачи перебора соединений таблиц,так как он требует сложных вычислений и значительных затрат ресурсов.
Сложность задачи обусловлена тем, что количество возможных соединений(бинарных) с n таблицами равно количеству бинарных дерьев 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 временя планирования становятся слишком велики. Эвристические методы не гарантируют оптимальность решения, так как находят локальное оптимальные планы, но не гарантируют глобальное оптимальное решение.
Современные подходы предлагают использовать машинное обучение на основе статистик, которые могут динамически меняться в процессе выполнения запроса. И создавать на их основе 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 проанализировать классические и современные 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.
- Изучить современные подходы, основанные на машинном обучении, и их применение в автоматическом выборе порядка соединений.
- Провести сравнительный анализ эффективности различных методов и выявить сценарии нагрузок, при которых каждый из них показывает наилучшие результаты.
- Выработать критерии для выбора оптимального метода планирования соединений в зависимости от типа запроса и структуры данных.

Исследование позволит оценить, какие подходы обеспечивают наилучшую п роизводительность 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 и в каком порядке сое. Существуют три типа соединений:

Nested Loop Join – выполняет вложенный перебор строк. Для каждой строки из левой таблицы проходится по всем строкам правой таблицы и ищет соответствующие строки по условию соединения. Пусть левая таблица занимает M страниц и имеет m кортрежей, правая N, n соответственно. Имеется варианты NLJ:

- **Наивный NLJ** – перебирает каждый кортеж левой таблицы, сравнивая с каждым 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 в буфере. 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 правой таблицы, проверяем условие с каждой строкой из буфера правой страницы, 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 отсортированных таблиц. Сложность $O(+M+N)$.

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 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 оптимазации, то есть перевода лоогического дерева в физическое

Задача выбора порядка соединений

С увеличением количества данных в СУБД увеличивается и время, необходимое для выполнения запросов. Сокращение времени выполнения запросов становится решающим фактором для повышения удобства и эффективности работы пользователей. СУБД преобразует запрос в набор планов. Каждый план представлен в виде дерева, узлами 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 [https://ruor.uottawa.ca/items/9898a486-a7ea-42af-a6d9-66b165a90e33] приводится пример разной стоимости для двух деревьев соединений.

Выбор в какой последовательности нужно соединить таблицы является задачей выбора порядка соединений. Различные планы запросов для одного и того же SQL 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. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый. Затем выбранный вариант разворачивается в полноценный план, который сможет использовать исполнитель. Мы примем, что уже получена стоимость каждого оператора сканирования.

Факторы влияющие на производительность

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

Размер отношений (кардинальность таблицы) – 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.

Кардинальность условия(селективнось 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 корреляцию данных, селективность соединения. Зная распределение, оптимизатор может делать предположения, как данных хранятся в отношениях: есть ли отсортированность, использовать последовательное сканирование или индекс для поиска.

Методы ускорения перебора соединений таблиц

СУБД применяют классические и современные 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 зачачтую имеют плохую ассимптотическую сложность.
- Эвристические методы, требуют меньщих затра ресурсов по сравнению с ДП, но находят приблизительно (локально) оптимальные планы.

Современные методы машинного обучения (ML) включают в себя:

- Обучение с подкреплением (Reinforcement Learning), позволяющее адаптивно находить оптимальный порядок соединений на основе исторических данных и дообучаться в процессе работы.
- Специальные модели СУБД, запоминающие часть истории запросов и использующие их для дообучения.

Классические подходы

Данные подходы в большинстве своём представляют алгоритмы динамического прогаммирования и эвристические.
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.
С другой стороны 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, DPhyp и LinDP++. А также эвристические алгоритмы GOO(Greedy Operator Ordering) и Geqo. Помимо этого разберём как работает планировщик в **PostgreSQL**.

DPsize [https://db.in.tum.de/teaching/ws2425/queryopt/main.pdf?lang=de]

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 они не коммутативны.
Изначально B 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.

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

Сложность DPsize[https://dsg.uwaterloo.ca/seminars/notes/Guido.pdf]:

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

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

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

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

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

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 [https://db.in.tum.de/teaching/ws2425/queryopt/main.pdf?lang=de]

Разбивает граф соединений на подграфы, строит оптимальный порядок для каждого компонента и соединяет их. Также не поддерживает работу с внешними соединениями. Перебирает всевозможные подмножества таблиц S. Разделение множество S на S_1 и S_2, так же как в DPsize.

```
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
```

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.

$$I^{\text{chain}}_{\text{DPsub}}(n) = 2^{n+2} - n^2 - 3n - 4$$

$$I^{\text{cycle}}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2$$

$$I^{\text{star}}_{\text{DPsub}}(n) = 2 \times 3^{n-1} - 2^n$$

$$I^{\text{clique}}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1$$

DPccp

[https://db.in.tum.de/teaching/ws2425/queryopt/main.pdf?lang=de][https://dsg.uwaterloo.ca/se

Пусть дан граф соединений G = (V, E). Введём понятие csg-cmp-pair (S1, S2) – в графе запроса выдели S_1, S_2 связные непересекающиеся подграфы, такие что $S_1 \in V$, $S_2 \in V/S_1$ существует между ними существует ребро(условие соединения).
Заметим, что если (S_1, S_2) - csg-cmp-, 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, перечисляемых нашей процедурой, не будет вычислено ни одной дублирующей csg-cmp-pair. Как следствие, мы должны позаботиться о том, чтобы наша процедура динамического программирования была полной: если применяемый нами бинарный оператор коммутативен, то процедура построения плана для $S_1 \cup S_2$ из планов для S_1 и S_2 должна учитывать коммутативность.
Однако это не представляет особой сложности. Очевидно, что для того, чтобы быть корректным, любой алгоритм динамического программирования должен учитывать все csg-cmp-pairs. Таким

образом, минимальное количество вызовов функции стоимости любого алгоритма динамического программирования в точности равно количеству csg-cmp-pairs для данного гиперграфа. Заметим, что число связных подграфов намного меньше числа csg-cmp-pairs. Теперь задача состоит в том, чтобы перечислить csg-cmp-pairs эффективно и в порядке, приемлемом для динамического программирования. 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.

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** $B[\{R_0, \ldots, R_{n-1}\}]$

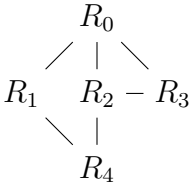
Сначала алгоритм инициализирует начальные значени $B[R_i] = R_i$. Затем перебираются такие подмножества вершин S_1 и S_2, что (S_1, S_2) – csg-cmp-pair и $B[S_1]$, $B[S_2]$ уже известны. Берутся оптимальные планы p_1 и p_2 от S_1 и S_2, затем они соединяются. Если стоимость получившегося плана больше B[S], то происходит замена. Ограничения:

- Не поддерживает внешние соединения, из-за разбиения на csg-cmp-pairs.
- Сильно зависит от эффективности поиска csg-cmp-pair.

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

- Эффективно пронумеровать все связные графы (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 узлы:
$V = R_0, \ldots, R_{n-1}$ – избавимся от повторного добавления (S_1, S_2) и (S_2, S_1). Пусть G = V, E — граф запроса. Введём множества $B_i = v_j | j <= i$ и функцию соседства
$N(V^{'}) = v^{'} | v \in V^{'} (v, v^{'}) \in E$



Algorithm EnumerateCsg

1: **for** $i = n - 1$ **to** 0 **do**
2: **emit** u_i;
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, S')
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
$\{4\}$	$\{0,1,2,3,4\}$	$\emptyset$	
$\{3\}$	$\{0,1,2,3\}$	$\{4\}$	$\{3,4\}$
$\{2\}$	$\{0,1,2\}$	$\{3,4\}$	$\{2,3\}$ $\{2,4\}$ $\{2,3,4\}$
$\{1\}$	$\{0,1\}$	$\{4\}$	$\{1,4\}$
$\rightarrow \{1,4\}$	$\{0,1,4\}$	$\{2,3\}$	$\{1,2,4\}$ $\{1,3,4\}$ $\{1,2,3,4\}$
$\{0\}$	$\{0\}$	$\{1,2,3\}$	$\{0,1\}$ $\{0,2\}$ $\{0,3\}$
$\rightarrow \{0,1\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,1,4\}$
$\rightarrow \{0,2\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,2,4\}$
$\rightarrow \{0,3\}$	$\{0,1,2,3\}$	$\{4\}$	$\{0,3,4\}$

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: EnumerateCmpRec$(G, \{u_i\}, X \cup N)$
9: **end for**

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

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

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

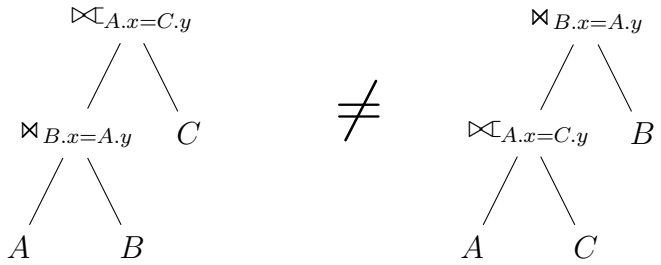
$$I^{\text{star}}_{\text{DPccp}}(n) = n2^n$$

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

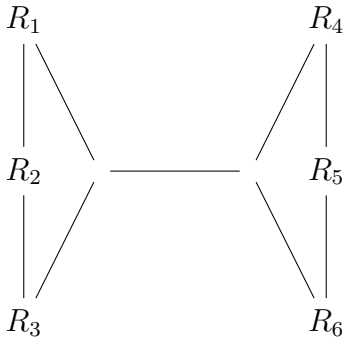
Можно заметить, что также как и DPsize и DPsub, DPccp не поддерживает внешние соединения. Однако он на каждом типе графа имеет меньшую сложность и имеет более сложную реализацию чем DPsize и DPsub.

DPhyp[https://15721.courses.cs.cmu.edu/spring2020/papers/20-optimizer2/p539-moerkotte.pdf]

Является улучшенной версией алгоритма DPccp для работы с внешними соединения, которые не всегда коммутативны, а значит не всегда допускают переупорядочивания. Это значит, что если мы сделаем невалидное переупорядочивание результат изменится.



При работе с внешними соединениями возникает понятие гиперграфа.
Гиперграф – пара (V, E), где V – непустове 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 узлы 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 гипреграф 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 узлы 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 фукнцию 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 из возрастающей части графа запроса.
- Как и основные csg, так и их cmp создаются путём рекурсивного обхода графа.
- Во время обходя графа, некоторые узлы запрещены для посещения, чтобы избежать дублирования csg-cmp-pair.
- Csg увеличиваются за счёт ребёр, ведущим к соседним узлам. Гиперрёбра интерпретируются как n к 1 рёбра: n рёбер с одной стороны ведут к одному узлу с другой стороны.

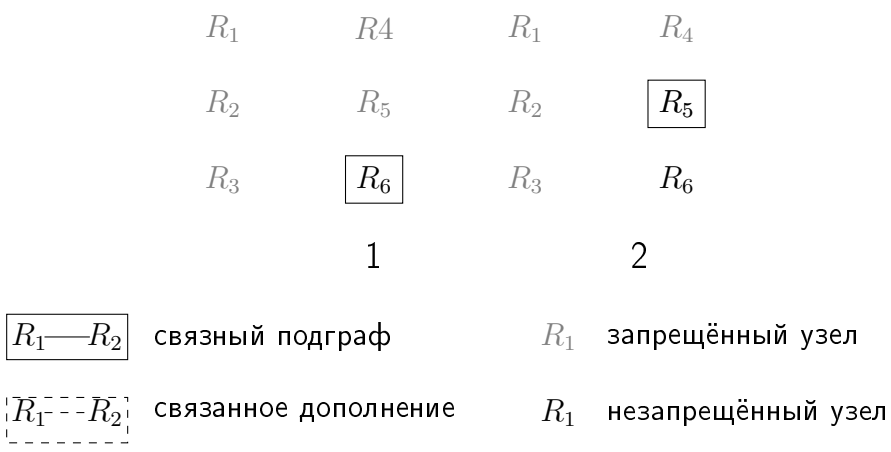
Алгоритм обходит граф в фиксированном порядке и рекурсивно производит более крупные связные подграфы. Но есть некоторые проблемы:

- Начальная сторона гиперребра может включать в себя несколько узлов, что усложняет построение соседства.
- Конечная сторона ребра может приводить сразу к нескольким узлам, что нарушает рекурсивный рост компонентов. Поэтому алгоритм выбирает конечный узел (1 в «n к 1»), начинает рекурсивный рост и использует таблицу DP для проверки, достигнуто ли правильное построение.

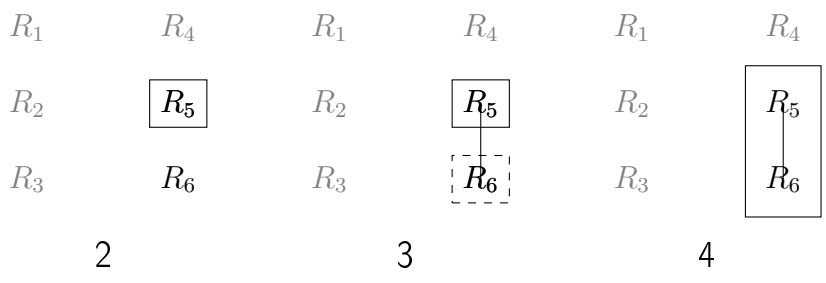
Весь алгоритм распределен по пяти подпрограммам. Подпрограмма верхнего уровня Solve инициализирует динамическую таблицу программирования с планами доступа для одиночных отношений, а затем вызывает EmitCsg и EnumerateCsgRec для каждого набора, содержащего ровно одно отношение. EnumerateCsgRec отвечает за перечисление связных подграфов. Для этого

она вычисляет окрестность и перебирает каждое ее подмножество. Для каждого такого подмножества S_1 она вызывает EmitCsg. Эта функция отвечает за нахождение подходящие дополнения. Для этого нужно вызвать EnumerateCmpRec, который рекурсивно перечисляет дополнения S_2 для найденного ранее связного подграфа S_1. Пара (S_1, S_2) является csg-cmp-pair. Для каждой такой пары вызывается программа EmitCsgCmp. Ее основная задача – рассмотреть план, построенный из планов для S_1 и S_2.

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



В первом цикле инициализируется таблица ДП с планами для одиночных отношений. Во втором цикле для каждого узла графа запросов в порядке убывания (согласно $\prec$) вызываются две подпрограммы `EmitCsg` и `EnumerateCsgRec`.
Алгоритм вызывает `EmitCsg`($\{v\}$) для отдельных вершин $v \in V$, чтобы сгенерировать 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** генерируется 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).

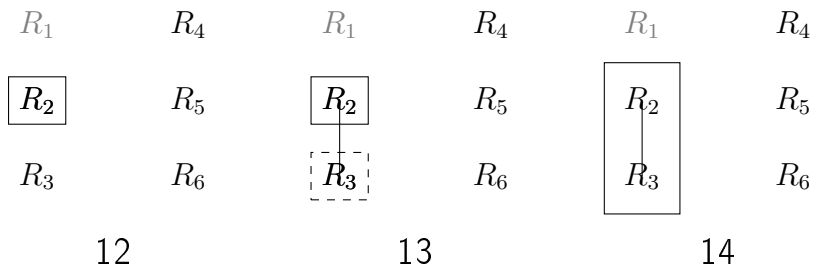


Чтобы избежать дублирования при перечислении, все узлы, которые упорядочены перед v согласно $<$, запрещены при рекурсивном расширении. Формально мы определяем это множество как $B_v = w|w < v \cup v$.

EnumerateCsgRec(S_1, X)

```
1: for each N ⊆ 𝒩(S1, X) : N ≠ ∅ do
2:     if dpTable[S1 ∪ N] then
3:         EmitCsg(S1 ∪ N)
4:     end if
5: end for
6: for each N ⊆ 𝒩(S1, X) : N ≠ ∅ do
7:     EnumerateCsgRec(S1 ∪ N, X ∪ 𝒩(S1, X))
8: end for
```

Цель `EnumerateCsgRec` — расширить заданное множество S_1, которое индуцирует связный подграф G, до большего множества с тем же свойством. Для этого рассматривается каждое непустое подходящее подмножество окрестности 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 окрестности мы вызываем `EnumerateCsgRec` так, что $S_1 \cup N$ может быть рекурсивно расширен.
Причина, по которой мы сначала вызываем `EmitCsg`, а затем `EnumerateCsgRec`, заключается в том, что для того, чтобы иметь последовательность перечислений, действительную для динамического программирования, сначала должны быть сгенерированы меньшие наборы.

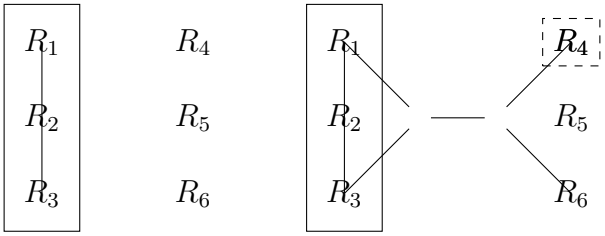


На шаге 12 был сгенерирован `Solve` на $S_1 = \{R_2\}$. Окрестность состоит только из $\{R_3\}$, так как R_1 находится в X (R_4, R_5, R_6 тоже не в X, так как не достижимы).
`EnumerateCsgRec` сначала вызывает `EmitCsg`, который создаст присоединяемое дополнение (шаг 13). Затем он проверяет $\{R_2, R_3\}$ на связность.
Соответствующая запись в `dpTable` была сгенерирована на шаге 13. Следовательно, этот тест проходит успешно, и $\{R_2, R_3\}$ далее обрабатывается рекурсивным вызовом `EnumerateCsgRec` (шаг 14).
Теперь расширение останавливается, так как окрестность $\{R_2, R_3\}$ пуста, поскольку $R_1 \in X$.
`EmitCsg` принимает в качестве аргумента непустое подходящее подмножество S_1 из V, которое индуцирует связный подграф. Его задача — сгенерировать все узлы S_2, такие, что (S_1, S_2) становится **csg-cmp-pair**.
Неудивительно, что узлы берутся из окрестности S_1. Все узлы, которые упорядочивались до наименьшего элемента в S_1 (охваченного множеством $B_{\min}(S_1)$), удаляются из окрестности, чтобы избежать дублирования перечислений [?].
Поскольку окрестность также содержит $\min(v)$ для гиперрёбер (u, v) с $|v| > 1$, то не гарантируется, что S_1 соединён с v. Чтобы избежать генерации ложных **csg-cmp-pairs**, `EmitCsg` проверяет их на связность.

EmitCsg(S_1)

1: X = $S_1 \cup B_{min(S_1)}$
2: N = $\mathcal{N}(S_1, X)$
3: **for** each $u \in N$ descending according to $<$ **do**
4: S_2 = u
5: **if** $\exists(u, v) \in E : u \subseteq S_1 \cap v \subseteq S_2$ **then**
6: EmitCsgCmp(S_1, S_2)
7: **end if**
8: EnumerateCmpRec(S_1, S_2, X)
9: **end for**

Однако каждый атомарный сосед может быть расширен до действительного дополнения S_2 к S_1. Поэтому перед вызовом `EnumerateCmpRec`, который выполняет это расширение, такая проверка не требуется.



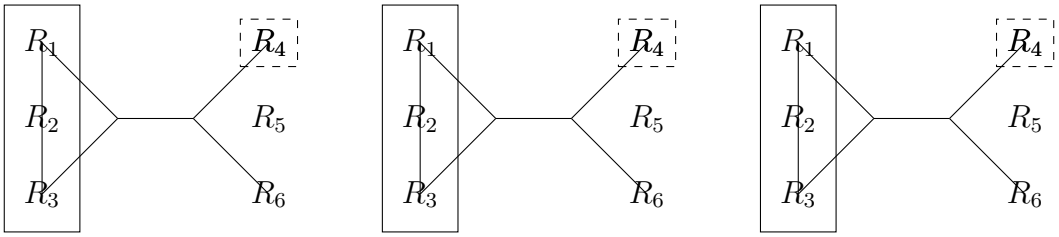
Посмотрите на шаг 20. Текущий набор $S_1 = \{R_1, R_2, R_3\}$, а $N = \{R_4\}$. Поскольку нет гиперребра, соединяющего эти два множества, вызов `EmitCsgCmp` не требуется.
Однако множество $\{R_4\}$ может быть расширено до допустимого дополнения, а именно $\{R_4, R_5, R_6\}$. Подходящее узловое семя дополнения является задачей вызова `EnumerateCmpRec` на шаге 21.

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**

`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 окрестность S_2.
Для каждого непустого подходящего подмножества N окрестности 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` ведёт себя очень похоже на `EnumerateCsgRec`.



21 22 23

Снова посмотрите на шаг 21. $S_1 = \{R_1, R_2, R_3\}$ и $S_2 = \{R_4\}$. $N = \{R_5\}$.
Множество $\{R_4, R_5\}$ индуцирует связный подграф. Он был вставлен в `dpTable` на шаге 6. 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 окрестностью является $\{R_6\}$. Множество $\{R_4, R_5, R_6\}$ приводит к связному подграфу.
Соответствующий тест через поиск в `dpTable` проходит успешно, так как соответствующая запись была сгенерирована на шаге 7. Вторая часть проверки также успешна, так как наш единственный истинный гиперребро соединяет это множество с S_1.
Следовательно, вызов `EmitCsgCmp` на шаге 23 происходит и генерирует 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 ∪ S_2
4:  p = ⋀_{(u_1,u_2)∈E, u_i⊆S_i} P(u_1, u_2)
5:  newplan = plan_1 ⋈_p plan_2
6:  if dpTable[S] = ∅ ∨ cost(newplan) < cost(dpTable[S]) then
7:      dpTable[S] = newplan
8:  end if
9:  newplan = plan_2 ⋈_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**-пару.
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 динамического программирования 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 гиперграниц, которые соединяют S_1 и S_2.
Затем планы строятся, и если они дешевле существующих, они сохраняются в `dpTable`.
Таким образом DPhyp, является относительно(DPsize, DPsub) эффективным алгоритмом. При этом позволяет обрабатывать внешние соединения.
Тем не менее, когда количество таблиц для соединения становится большим(например >15) все алгоритмы динамического программирования 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). Для решения таких задач прибегают к использованию двух техник: эвристические алгоритмы(будут описаны позже) и линеаризацию пространства поиска для алгоритмов динамического программирования (современный подход). [https://db.in.tum.de/ radke/papers/lindp++.pdf]
Данный подход являяется развитием линеаризованного DP(LinDP)[https://db.in.tum.de/ radke/papers/hugejoins.pdf] для работы с гиперграфами. Идея linDP состоит в том, сначала выбрав хороший (в идеале оптимальный) относительный порядок отношений, а затем использовать полиномиальный по времени шаг DP для построения оптимального кустового дерева для этого относительного порядка. Изначальный порядок строится алгоритмом IK/KBZ[IK84, KBZ86], который строит левостороннее дерево порядка с близким к оптимальному порядку за O(n^2). Однако алгоритм IK/KBZ не поддерживает внешние соединения и графы с запросов с циклами. Во втором случае строится минимальное остовное дерева, в котором убирают рёбра с наименьшей селективностью. На следующем шаге за O(n^3) строится близкое к оптимальное кустовое дерево.
Для начала введём важные определения: Пусть дан граф $G = (V, E)$, и начальное отношение R_k. Построим ориентированный **граф предшествования** $G_k^P = (V_k^P, E_k^P)$ с корнем в R_k:

- Выберем R_k как корневой узел в G_k^P, $V_k^P = R_k$
- Пока $|V_k^P| < |V|$, выберем $R_i \in V \setminus V_k^P$ такой, что $\exists R_j \in V_k^P : (R_j, R_i) \in E$. Добавим R_i в V_k^P и добавим (R_j, R_i) в E_k^P

Граф предшествования по сути вводит упорядочивание соединений.
Пусть A и B — две последовательности, а V и U — две непустые последовательности. Мы говорим, что функция стоимости C обладает *свойством перестановки смежных последовательностей* (ASI-свойством), если и только если существует функция T и ранговая функция, определённая как

$$\text{rank}(S) = \frac{T(S) - 1}{C(S)}$$

такое, что выполняется следующее:

$$C(AUVB) \leq C(AVUB) \iff \text{rank}(U) \leq \text{rank}(V)$$

если $AUVB$ и $AVUB$ удовлетворяют ограничениям предшествования, заданным некоторым графом предшествования.

TODO

Все расмотренные алгоритмы порождали планы в виде ветвистых деревьев, так как они в отличие от односторонних деревьев позволяют использовать преимущества параллелизма. Исполнение можно распараллелить, если они не зависят друг от друга, т.е соединения A и B можно выполнить параллельно, если на любом пути от корня до таблицы нету одновременно A и B. Такого нельзя добиться на односторонних деревьях.

Рассмотрим теперь эвристические методы планирования запросов. Применяемые когда количество таблиц в запросе велико.

GOO(Greedy Operator Order)

[https://db.in.tum.de/teaching/ws2425/queryopt/main.pdf?lang=de][https://dsg.uwaterloo.ca/se

GOO $[https://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisAllam19.pdf]$ – жадный алгоритм, который выбирает локально(не рассматривает все варианты порядка соединений) 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.

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 (mutation)** – 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** 1–5 определённое количество итераций, и выбор самого приспособленного из выживших.

Рассмотрим теперь реализацию работы планировщика в **PostgreSQL**.