Аннотация

Работа посвящена разработке новых эвристических методов планирования запросов в реляционных СУБД. В литературе известно достаточно много алгоритмов планирования, но в индустриальных применениях известны лишь базовые подходы, опирающиеся на динамическое программирование, жадные и генетические алгоритмы и их простейшие эвристические комбинации, позволяющие выбирать тот или иной алгоритм в зависимости от числа таблиц в запросе. Данная работа преследует цель получить глубокое понимание того, как разные алгоритмы возможно комбинировать в ходе планирования запроса, делая переключение между алгоритмами в зависимости от топологии соединений и выделенного временного бюджета на планирование. Ожидаемым результатом работы является решение оптимизационной задачи планирования с помощью новых эвристик, реализация решения в виде модификации планировщика в ядре реляционной СУБД с открытым кодом и экспериментальная оценка на известных индустриальных бенчмарках.

Введение

Современные СУБД работают с большим объёмом информации и транзакций, используя для ввода запросов язык SQL. С ростом количества данных увеличивается и время, необходимое для выполнения запросов. Сокращение этого времени выполнения запросов становится важным фактором для удобства и эффективности СУБД.

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-трудной задачи выбора оптимального порядка соединений таблиц, так как оно требует сложных вычислений и значительных затрат ресурсов операционной системы. В процессе работы оптимизатора, решается вопрос какой тип соединения использовать, как и в каком порядке соединить таблицы. Запрос преобразуется в набор планов.

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. Соединение принимает два аргумента, назовём первый аргумент левым, второй правым. Каждый аргумент это таблица, либо результат соединения нескольких таблиц.

Отношение – либо единичная таблица, либо результат соединения нескольких таблиц.

Виды соединений таблиц:

1. **Inner join** - возвращает только те строки, для которых условие связи выполняется в обоих аргументах.
2. **Outer join** - возвращает все строки из одной или обоих аргументов, дополняющее отсутствующие совпадения пустыми значениями. Существует три вида внешних соединений.
   (a) **LEFT OUTER JOIN** - возвращает все строки из левого аргумента и совпадения из правого.

(b) **RIGHT OUTER JOIN** - возвращает все строки из правого аргумента и совпадения из левого.

(c) **FULL OUTER JOIN** - возвращает объединение всех строк из обоих аргументов.

3. **SEMI JOIN** - возвращает строки из левого аргумента по существованию хотя бы одного совпадения в правом аргументе.

4. **Anti join** - это операция, обратная SEMI JOIN, которая возвращает строки из левого аргумента только при отсутствии совпадений в правом.

5. **Cross join** - соединяет каждую строку левого аргумента с каждой строкой правого. Выполняет соединение без условия.

Оптимизатор СУБД использует стоимостную модель (cost model), которая преобразует прогнозы использования различных ресурсов в одно число. **Стоимость**. Ключевые компоненты, которые учитываются, это:

1. Стоимость чтения/записи одной единицы данных, находящихся на ПЗУ.

2. Стоимость чтения/записи одной единицы данных, находящихся на ОЗУ.

3. Стоимость обработки одной единицы данных ЦПУ.

4. Стоимость передачи одной единицы данных по сети.

5. Стоимость использования памяти

Каждый план можно представить в виде дерева, вершинами в котором являются отношения. Рёбра — условия соединения отношений. При этом операция соединения не ассоциативна по стоимости, то есть $(R1 \bowtie R2) \bowtie R3 \neq R1 \bowtie (R2 \bowtie R3)$.

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.

Комбинаторная природа задачи видна через простую оценку. Если соединения выполнять бинарно, форма плана задаётся двоичным деревом с n листьями(таблицами). Таких структур C_{n-1}, где C_k — число Каталана; асимптотически $C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$. Даже без учёта перестановок самих отношений это уже экспоненциальный рост количества возможных планов.

В работе запрос рассматривается как (гипер)граф соединений. Вершины — отношения (таблицы), рёбра — условия соединений. Рёбра, затрагивающие более двух таблиц – гипер-рёбра. Ситуацию осложняют внешние соединения (`OUTER JOIN`): в общем случае они семантически не ассоциативны и не коммутативны, следовательно, ограничивают допустимые переупорядочивания.

Можно заметить, что гиперграф запроса можно разбить на различные топологии: цепи, циклы, звёзды, плотные графы. Количество возможных планов у двух топологий (с обычными рёбрами и без внешних соединений) от одинакового числа таблиц может отличаться в несколько раз, при увеличении количества таблиц разница может вырасти до нескольких порядков.

На практике индустриальные СУБД комбинируют подходы. Для малого числа таблиц применяют динамическое программирование (перебор подмножеств с запоминанием лучших частичных планов). При росте количества таблиц переключаются на эвристики: жадные стратегии, локальные улучшения, генетические алгоритмы и их простые сочетания. У такого подхода есть существенный недостаток: количество возможных планов коррелирует с числом таблиц, но линейная зависимость отсутствует. Также в индустрии есть спрос на тарифицируемое планирование и исполнение запросов: хотелось бы иметь возможность задать ограниченный бюджет на обработку запроса.

В данной работе предлагаются методы адаптивного планирования запросов, которые включают в себя:

1. Различные эвристики для планирования запроса.
2. Критерии для выбора эвристик для планирования запроса.
3. Критерии для динамического переключения между эвристиками во время планирования.
4. Реализацию динамическое переключение между эвристиками в зависимости от критериев.

Предшествующие работы

Задача выбора порядка соединений

В статье "On the Optimal Nesting Order for Computing N-Relational Joins" — Ibaraki & Kameda (1984) формализуется задача выбора оптимального порядка соединения n отношений, при использовании вложенного цикла для сканирования таблиц и результатов соединений. **Оптимальный вложенный порядок**. Для заданного набора отношений и предикатов соединения выводится формула ожидаемого числа чтений страниц как функция от порядка соединяемых отношений, и требуется найти порядок, минимизирующий это значение. Авторы прямо отмечают, что задача нахождения минимального значения этой функции NP-трудная.
Задача выбора оптимального вложенного порядка принимает на вход:

1. Статистики для формулы ожидаемого числа чтений страниц.
2. Описание запроса (граф соединений).
3. Пороговое значение B — существует ли такой порядок вложенности отношений, что число чтений страниц $\leq B$?

Принадлежность NP

Если мы угадали порядок отношений π, то значение функции числа чтений страниц для π вычисляется полиномиально по размеру входа (это просто вычисление выражения по заданным параметрам). Значит, решение можно проверить за полиномиальное время, тогда задача принадлежит NP.

NP-трудность

В доказательстве NP-трудности авторы сводят к этой задаче классическую NP-полную задачу поиска полного графа (Робин Карп, 1972): по графу $G = (V, E)$ и числу k строится экземпляр запроса (в терминах отношений и условий соединения) и подбираются параметры стоимости так, что существует порядок с числом чтений страниц $\leq B$ тогда и только тогда, когда в G есть полный граф размера k.

Пространство поиска планов

Пусть даны два отношения R1 и R2, и дан результат их соединения $(R1 \bowtie R2)$. Назовём R1 - левым сыном, R2 - правым сыном, а результат соединения родителем. Тогда получится двоичное дерево. Каждому возможному порядку соединения исходного набора таблиц сопоставим двоичное дерево.

[p493: On the Correct and Complete Enumeration of the Core Search Space (2013)]

Для заданного начального дерева **Пространство поиска планов** — это множество всех различных порядков, которые можно получить, применяя к начальному плану последовательность из корректных преобразований (они сохраняют эквивалентность результатов, т.е семантическую корректность).

Виды корректных преобразований:

1. **Коммутативное**: $(R \bowtie_{RS} S) = (S \bowtie_{SR} R)$
2. **Ассоциативное**: $(R \bowtie_{RS} S) \bowtie_{RT,ST} T = R \bowtie_{RS,RT} (S \bowtie_{ST} T)$
3. **Левоассоциативное**: $(R \bowtie_{RS} S) \bowtie_{RT,ST} T = (R \bowtie_{RT} T) \bowtie_{RS,ST} S$
4. **Правоассоциативное**: $R \bowtie_{RS,RT} (S \bowtie_{ST} T) = S \bowtie_{RS,ST} (R \bowtie_{RT} T)$

Жадный подход к планированию

В статье "A New Heuristic for Optimizing Large Queries" — Fegaras (1998) предлагается эвристический способ построения оптимального порядка, путём ухода от экспоненциального перебора порядков соединений, строя порядок соединений жадно снизу-вверх:

1. На каждом шаге выбирать такое соединение, что даёт минимальный размер промежуточного результата, с учётом селективностей предикатов.
2. Соединять выбранные поддеревья.
3. Обновлять граф запроса новыми оценками размеров/селективностей.

Algorithm 1 GOO($\{R_1, \ldots, R_n\}$, *weight*(T_1, T_2))

Require: set of relations to be joined
Ensure: join tree
1: *Trees* $\leftarrow \{R_1, \ldots, R_n\}$
2: **while** $|$*Trees*$| \neq 1$ **do**
3: find $T_i, T_j \in$ *Trees* such that $i \neq j$ and *weight*(T_i, T_j) is minimal among all pairs of trees in *Trees*
4: *Trees* $\leftarrow$ *Trees* $\setminus \{T_i\}$
5: *Trees* $\leftarrow$ *Trees* $\setminus \{T_j\}$
6: *Trees* $\leftarrow$ *Trees* $\cup \{T_i \bowtie T_j\}$
7: **end while**
8: **return** the (only) tree contained in *Trees*

Преимущества:

1. Эвристика строит ветвистые деревья, а не только левосторонние, что позволяет параллельно исполнять дерево плана.
2. Полиномиальность по времени планирования $O(n^3)$, где n — число начальных отношений, вместо экспоненциального времени у подхода динамического программирования.

Недостатки:

1. Нет гарантий оптимальности: локально лучший шаг может загнать в глобально плохой порядок, классическая проблема жадных алгоритмов.
2. Зависимость от качества оценок кардинальностей/селективностей: если оценки ошибочны, жадный подход резко деградирует.

В этой статье также показывается связь с работой Ibaraki & Kameda (1984). Так как в общем виде задача выбора оптимального порядка соединения отношений NP-трудная, то планирование больших запросов требует больших вычислительных ресурсов, следовательно, нужны эвристики.

В современных СУБД самый распостранённый подход к планированию это **динамическое программирование**. Алгоритмы динамического программирования — это методы решения задач оптимизации, которые разбивают исходную задачу на набор перекрывающихся подзадач и используют принцип оптимальности: оптимальное решение строится из оптимальных решений подзадач. Вместо повторного пересчёта результатов они их запоминают, снижая асимптотическую сложность по сравнению с полным перебором. Такие алгоритмы особенно эффективны, когда пространство решений можно параметризовать состояниями (например, подмножествами, интервалами, путями в графе). Цена за ускорение — рост памяти и экспоненциальное число состояний в худшем случае.

Динамическое программирование по размеру подпланов (DPsize)

В учебнике "Building Query Compilers" — Moerkotte (2025) алгоритм DPsize рассматривается как вариант динамического программирования для построения оптимального ветвистого (bushy) дерева соединений без декартовых произведений при

условии, что граф соединений запроса связный. Алгоритм перечисляет планы в порядке возрастания размера подпланов (числа отношений внутри подплана), что является обобщением схемы DP-Linear-1 на ветвистые деревья.

DPsize поддерживает таблицу $BestPlan(S)$, сопоставляющую каждому множеству отношений S лучший (минимальной стоимости) найденный план, и строит планы снизу-вверх:

1. Инициализация: для каждого отношения R_i задаётся $BestPlan(\{R_i\}) = R_i$.
2. Для размера плана $s = 2, \ldots, n$ (по возрастанию) перебираются разбиения $s = s_1 + s_2$, где $1 \leq s_1 \leq \lfloor s/2 \rfloor$.
3. Для всех множеств S_1, S_2, уже имеющихся в $BestPlan$, таких что $|S_1| = s_1$, $|S_2| = s_2$, проверяется:
   (a) $S_1 \cap S_2 = \varnothing$ (подпланы не перекрываются),
   (b) S_1 соединено с S_2 хотя бы одним предикатом.
4. Если проверки пройдены, строится кандидат $CurrPlan = CreateJoinTree(BestPlan(S_1), BestPlan(S_2))$ и обновляется $BestPlan(S_1 \cup S_2)$, если кандидат дешевле.

1: **Input:** A set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
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** $1 < s \leq n$ **ascending do**
8: $\quad$ **for each** $S_1, S_2 \subset R$ **such that** $|S_1| + |S_2| = s$ **do**
9: $\quad\quad$ **if** (**not** cross products $\wedge \neg S_1$ connected to S_2) $\vee$ $(S_1 \cap S_2 \neq \emptyset)$ **then**
10: $\quad\quad\quad$ **continue**
11: $\quad\quad$ **end if**
12: $\quad\quad$ $p_1 \leftarrow B[S_1], p_2 \leftarrow B[S_2]$
13: $\quad\quad$ **if** $p_1 = \epsilon$ **or** $p_2 = \epsilon$ **then continue**
14: $\quad\quad$ **end if**
15: $\quad\quad$ $P \leftarrow$ CreateJoinTree(p_1, p_2)
16: $\quad\quad$ **if** $B[S_1 \cup S_2] = \epsilon$ **or** $C(B[S_1 \cup S_2]) > C(P)$ **then**
17: $\quad\quad\quad$ $B[S_1 \cup S_2] \leftarrow P$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: **end for**

Преимущества:

1. Оптимальность (в рамках выбранной стоимостной модели и класса ветвистых деревьев без декартовых произведений): DPsize перебирает все допустимые склейки подпланов и сохраняет лучший результат для каждого множества S.
2. Структурированное перечисление снизу-вверх по размеру подпланов: удобно для реализации и естественно соответствует принципу оптимальности динамического программирования.

3. Зависимость времени от топологии: для цепей и циклов автор приводит полиномиальные формулы числа внутренних проверок (четвёртая степень по n), что объясняет практическую применимость DPsize на простых топологиях (цепочка, цикл).

Недостатки:

1. Худший случай экспоненциален: для звезды и полного графа число комбинаций резко возрастает.

2. Далёк от нижней границы по перебору: в книге подчёркнуто, что DPsize (как и DPsub) перебирает существенно больше, чем необходимый минимум, что и мотивирует использовать DPccp.

Сложность DPsize от числа отношений n:

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

$$IDPsize_{\text{cycle}}(n) = \begin{cases} \frac{1}{4}\left(n^4 - n^3 - n^2\right), & n \text{ even}, \\ \frac{1}{4}\left(n^4 - n^3 - n^2 + n\right), & n \text{ odd}. \end{cases}$$

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

$$q(n) = n \cdot 2^{2n-1} - 5 \cdot 2^{n-3} + \frac{1}{2}(n^2 - 5n + 4).$$

$$IDPsize_{\text{clique}}(n) = \begin{cases} 2^{2n-2} - 5 \cdot 2^{n-2} + \frac{1}{4}\binom{2n}{n} - \frac{1}{4}\binom{n}{n/2} + 1, & n \text{ even}, \\ 2^{2n-2} - 5 \cdot 2^{n-2} + \frac{1}{4}\binom{2n}{n} + 1, & n \text{ odd}. \end{cases}$$

Динамическое программирование по подмножествам (DPsub)

В “Building Query Compilers” — Moerkotte DPsub описывается как вариант динамического программирования для построения оптимального ветвистого (bushy) дерева соединений без декартовых произведений. В отличие от DPsize, где планы строятся по возрастанию размера подпланов, DPsub перебирает все непустые подмножества исходного множества отношений и для каждого подмножества строит лучший план.

```
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
```

Преимущества:

1. Эффективен на плотных пространствах поиска (например, звезда/клика): в таких графах больше связных подмножеств и больше разбиений $S = S_1 \cup S_2$, проходящих проверки, поэтому доля холостых итераций меньше; на этих топологиях DPsub начинает выигрывать у DPsize.

2. Оптимальность (в рамках выбранной стоимостной модели и класса ветвистых деревьев без декартовых произведений): DPsub перебирает все допустимые разбиения S на две части и сохраняет лучший план для каждого S.

Недостатки:

1. Большое количество непройденных проверок для простых топологий (цепь/цикл): DPsub перебирает все подмножества S, но значительная часть из них несвязна, а для связных S большая доля разбиений (S_1, S_2) не проходит проверки связности/наличия предиката для соединения.

2. Далёк от теоретической нижней границы: для большинства топологий число проверок во внутреннем цикле на порядки больше числа количества пар связных подграфов (#ccp), что служит мотивацией перехода к DPccp.

Сложность DPsub от числа отношений n:

$$IDPsub_{\text{chain}}(n) = 2^{n+2} - n^2 - 3n - 4, \qquad IDPsub_{\text{cycle}}(n) = n \cdot 2^n + 2^n - 2n^2 - 2,$$

$$IDPsub_{\text{star}}(n) = 2 \cdot 3^{n-1} - 2^n, \qquad IDPsub_{\text{clique}}(n) = 3^n - 2^{n+1} + 1.$$

Динамическое программирование по csg-cmp-парам (DPccp)

Пусть дан граф соединений $G = (V, E)$. Введём понятие csg-cmp-pair (S_1, S_2) — в графе запроса выделим S_1, S_2 — связные непересекающиеся подграфы, такие что $S_1 \subseteq V$, $S_2 \subseteq V \setminus S_1$ (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.

В [Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Product] предлагается алгоритм DPccp как улучшение DPsize/DPsub для построения оптимального ветвистого дерева соединений без декартовых произведений. Мотивация следующая: DPsize и DPsub тратят много итераций внутреннего цикла на проверки, которые часто не проходят, и их объём работы может быть существенно больше теоретической нижней границы, задаваемой числом #ccp.
DPccp поддерживает таблицу $BestPlan(S)$ и вместо перебора всех разбиений подмножеств рассматривает ровно csg-cmp-pairs:

1. Инициализация: $BestPlan(\{R_i\}) = R_i$ для всех R_i.
2. Перебор всех csg-cmp-pairs (S_1, S_2); положим $S = S_1 \cup S_2$.
3. Для текущей пары берутся $p_1 = BestPlan(S_1)$, $p_2 = BestPlan(S_2)$, строится кандидат $CurrPlan = CreateJoinTree(p_1, p_2)$ и обновляется $BestPlan(S)$, если кандидат дешевле.
4. Так как процедура перечисления генерирует только одну ориентацию пары, алгоритм дополнительно учитывает коммутативность соединения, пробуя $CreateJoinTree(p_2, p_1)$.

Algorithm 2 DPccp

Require: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Ensure: an optimal bushy join tree

for all $R_i \in R$ **do**
    $\texttt{BestPlan}(\{R_i\}) \leftarrow R_i$
end for
for all csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ **do**
    $\texttt{InnerCounter} \leftarrow \texttt{InnerCounter} + 1$
    $\texttt{OnoLohmanCounter} \leftarrow \texttt{OnoLohmanCounter} + 1$
    $p_1 \leftarrow \texttt{BestPlan}(S_1)$
    $p_2 \leftarrow \texttt{BestPlan}(S_2)$
    $\texttt{CurrPlan} \leftarrow \texttt{CreateJoinTree}(p_1, p_2)$
    if $\texttt{cost}(\texttt{BestPlan}(S)) > \texttt{cost}(\texttt{CurrPlan})$ **then**
        $\texttt{BestPlan}(S) \leftarrow \texttt{CurrPlan}$
    end if
    $\texttt{CurrPlan} \leftarrow \texttt{CreateJoinTree}(p_2, p_1)$
    if $\texttt{cost}(\texttt{BestPlan}(S)) > \texttt{cost}(\texttt{CurrPlan})$ **then**
        $\texttt{BestPlan}(S) \leftarrow \texttt{CurrPlan}$
    end if
end for
$\texttt{CsgCmpPairCounter} \leftarrow 2 \cdot \texttt{OnoLohmanCounter}$
return $\texttt{BestPlan}(\{R_0, \ldots, R_{n-1}\})$

Преимущества:

1. Достижение нижней границы перебора: DPccp рассматривает ровно ccp; в тексте подчёркивается, что это является нижней границей для DP.

Недостатки:

1. Худший случай всё равно экспоненциален: на полных графах число #ccp очень быстро возрастает, и DPccp также становится очень дорогим.
2. Сложность реализации: нужно эффективно перечислять ccp без дубликатов и в порядке, корректном для DP, чтобы перед (S_1, S_2) уже были рассмотрены все непустые подмножества.

Сложность DPccp от числа отношений n:

$$IDPccp_{\text{chain}}(n) = n^3, \qquad IDPccp_{\text{cycle}}(n) = n^3, \qquad IDPccp_{\text{star}}(n) = n^2 2^n, \qquad IDPccp_{\text{clique}}(n) = 3^n.$$

Алгоритм DPccp перечисляет csg-компоненты S_1 (связные подмножества) в порядке, согласованном с DP, используя BFS-нумерацию вершин и запрет на включение вершин с меткой меньше стартовой, чтобы не порождать дубликаты. Для каждого S_1 процедура `EnumerateCmp` строит вторую часть пары S_2 только внутри дополнения $V \setminus S_1$: стартует с одиночных вершин из окрестности $N(S_1)$ и затем рекурсивно расширяет их, добавляя подмножества соседей уже построенного S_2 (то есть всегда растит связный компонент). Параметр исключения X выбирается так, чтобы S_2 не содержал вершин с меткой меньше любой вершины из S_1 (условие порядка $min(S_1) < min(S_2)$), что устраняет симметричные дубликаты (S_1, S_2) / (S_2, S_1). Из-за старта из $N(S_1)$ и связного расширения гарантируется смежность S_2 к S_1 (есть ребро между компонентами), поэтому перечисляются ровно допустимые csg-cmp-пары без кросс-продуктов.

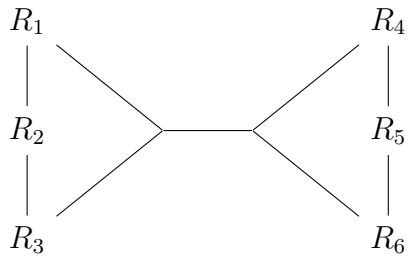

Рис. 1: Sample hypergraph

DPhyp: динамическое программирование на гиперграфах

[Dynamic Programming Strikes Back] DPccp эффективен для *простых* графов соединений (бинарные предикаты) и inner join: он перечисляет ровно csg-cmp-пары и тем самым близок к нижней границе перебора. Однако реальные запросы содержат *сложные предикаты*, связывающие сразу несколько отношений, и *non-inner joins* с ограничениями на перестановки. DPhyp обобщает DPccp на эти случаи за счёт представления запроса *гиперграфом* и того же принципа перечисления "осмысленных" склеек.

Гиперграф и csg-cmp-пары

Запрос представим гиперграфом $H = (V, E)$, где V — отношения, а гиперребро $e = (U, W)$ связывает две непересекающиеся группы $U, W \subseteq V$, $U \cap W = \emptyset$ (частный случай $|U| = |W| = 1$ даёт обычное ребро). Подмножество $S \subseteq V$ называется *csg* (connected subgraph), если индуцированный подграф связан. Для csg-множества S_1 множество $S_2 \subseteq V \setminus S_1$ называется *cmp* (connected complement), если индуцированный подграф на S_2 связан. Пара (S_1, S_2) есть *csg-cmp-пара*, если S_1 — csg, S_2 — cmp для S_1, и существует гиперребро $(U, W) \in E$ такое, что $U \subseteq S_1$ и $W \subseteq S_2$ (т.е. между компонентами есть предикат соединения).

Идея перечисления

DPhyp перечисляет только csg-cmp-пары, избегая массовых "пустых" проверок, характерных для DPsize/DPsub. Вводится линейный порядок на отношениях $\prec$ и правило уникальности: рассматриваются только пары с $\min(S_1) \prec \min(S_2)$ (симметричные дубликаты не порождаются). Компоненты строятся рекурсивным расширением через *окрестность* $N(S, X)$ — разрешённые узлы, "достижимые" из S по гиперребрам, исключая запрещённые X. Для гиперребра (U, W), где $U \subseteq S$, в окрестность добавляется канонический вход $\min(W)$: это позволяет корректно инициировать построение второй стороны гиперребра и затем достраивать недостающие узлы.

Пусть есть цепочка $R_1 - R_2 - R_3$, цепочка $R_4 - R_5 - R_6$ и гиперребро между $\{R_1, R_2, R_3\}$ и $\{R_4, R_5, R_6\}$. Тогда $(S_1, S_2) = (\{R_1, R_2, R_3\}, \{R_4, R_5, R_6\})$ — валидная csg-cmp-пара: обе стороны связны и соединяемы гиперребром. Характерная ситуация: при $S_1 = \{R_1, R_2, R_3\}$ окрестность даёт вход $\min(\{R_4, R_5, R_6\}) = R_4$, и процедура `EnumerateCmpRec` достраивает S_2 до полной тройки, после чего пара становится соединяемой.

Псевдокод DPhyp

```
Solve():
  for v in V: dp[{v}] = scan(v)
  for v in V (по убыванию ⊠):
    EmitCsg({v})
    EnumerateCsgRec(S1={v}, X=Bv)     // Bv = {w | w ⊠ v} ⊠ {v}
  return dp[V]

EnumerateCsgRec(S1, X):
  for each nonempty N ⊠ N(S1, X):
    if connected(S1 ⊠ N):
      EmitCsg(S1 ⊠ N)
      EnumerateCsgRec(S1 ⊠ N, X ⊠ N(S1, X))

EmitCsg(S1):
  X = S1 ⊠ Bmin(S1)
  for v in N(S1, X) (по убыванию ⊠):
    S2 = {v}
    if joinable(S1, S2): EmitCsgCmp(S1, S2)
    EnumerateCmpRec(S1, S2, X)

EnumerateCmpRec(S1, S2, X):
  for each nonempty N ⊠ N(S2, X):
    if connected(S2 ⊠ N) and joinable(S1, S2 ⊠ N):
      EmitCsgCmp(S1, S2 ⊠ N)
    EnumerateCmpRec(S1, S2 ⊠ N, X)

EmitCsgCmp(S1, S2):
  S = S1 ⊠ S2
  p = predicates_between(S1, S2)      // из всех гиперребер, пересекающих разрез
  dp[S] = min(dp[S], join(dp[S1], dp[S2], p))  // + вариант перестановки сторон
```

Для non-inner joins порядок перестановок ограничен семантикой, поэтому часть разбиений заведомо невалидна. Идея статьи: закодировать такие ограничения в структуре гиперграфа (добавляя "ограничивающие" гиперребра/предикаты), чтобы алгоритм DP перечислял только допустимые комбинации. После этого DPhyp можно применять без изменения ядра алгоритма: ограничения автоматически проявляются как отсутствие соединяемых csg-cmp-пар. Практический эффект — резкое уменьшение пространства поиска по сравнению с DP, который проверяет валидность перестановок на каждом шаге.

Генетический подход

В работе Bennett, Ferris, Ioannidis (1991) предлагается генетический алгоритм (GA) для оптимизации запросов соединения как альтернатива классическому динамическому программированию типа System R. Запрос рассматривается как дерево выполнения соединений (join processing tree), а качество решения определяется стоимостной моделью (cost) и переводится в функцию приспособленности, которую GA максимизирует (обычно беря отрицание стоимости и масштабируя). Алгоритм

поддерживает популяцию планов кандидатов, итеративно применяя селекцию, кроссовер и мутацию.

Ключевые идеи

1. Два пространства поиска (strategy spaces):
   (a) L — только left-deep планы (линейные деревья).
   (b) A — более общее пространство, включающее bushy планы (ветвистые деревья).

   Мотивация: оптимальный план часто лежит вне L, поэтому для выигрыша по качеству требуется искать в A.

2. Кодирование планов (chromosome encoding):
   (a) Для L: хромосома — упорядоченный список генов вида (relation, join method).
   (b) Для A: хромосома — упорядоченный список генов, где каждый ген соответствует конкретному соединению из графа запроса, вместе с методом соединения и информацией о порядке (outer/inner).

3. Декодирование и запрет декартовых произведений: декодирование строит дерево снизу-вверх и обеспечивает выполнимость; планы, порождающие декартовы произведения, штрафуются (например, бесконечной стоимостью), а генератор начальной популяции для left-deep старается создавать хромосомы без кросс-продуктов.

4. Локальная селекция (local neighborhood GA): вместо глобальной селекции используется схема локальной окрестности (например, ring6), где отбор партнёров для скрещивания происходит внутри локального окружения хромосомы.

5. Операторы поиска:
   (a) Мутации: (i) случайная смена метода соединения; (ii) локальная перестановка (swap) соседних генов.
   (b) Кроссовер: два оператора — M2S (modified two swap) и CHUNK (перенос случайного непрерывного фрагмента генов).

Преимущества

1. Масштабируемость для больших запросов: GA не требует хранения DP-таблиц экспоненциального размера и остаётся применимым там, где System R становится непрактичным из-за памяти.

2. Возможность улучшать планы относительно left-deep оптимума: поиск в A позволяет находить ветвистые планы, которые по стоимости могут быть лучше лучшего left-deep плана.

3. Хорошая параллелизуемость: популяционная природа GA естественно переносится на параллельную архитектуру с небольшими коммуникационными затратами.

Недостатки

1. Нет гарантий оптимальности и стабильности: по мере роста размера запроса качество решений и устойчивость сходимости ухудшаются из-за резкого роста пространства стратегий.

2. Чувствительность к параметрам: качество зависит от размера популяции, выбора операторов кроссовера/мутации и схемы селекции; неудачные настройки дают деградацию.

3. Затраты на оценку приспособленности: нужно многократно вычислять стоимость планов для большого числа хромосом, что может доминировать во времени оптимизации.

Использование графовых топологий при оценке эвристик

В работе Allam (2018) порядок соединений рассматривается через граф запроса: вершины — отношения, рёбра — предикаты соединения, веса рёбер — селективности. Автор использует типовые топологии (цепь, цикл, звезда, полный граф) как контролируемые модели различных классов графов соединений, чтобы сравнить поведение алгоритмов на структурах с разной плотностью рёбер и диаметром. Отдельно подчёркивается, что эффективность оптимизаторов зависит от структуры графа. Например, цепь и звезда с одинаковым числом таблиц имеют разное время планирования.

Подход: жадная эвристика и сравнение по топологиям

Основной исследуемый алгоритм — жадный подход, описанный выше.

Ключевые идеи оценки

1. Сравнить жадный подход и динамическое программирование по двум метрикам: (i) стоимость найденного плана, (ii) время оптимизации (runtime).

2. Выполнить сравнение на пяти графах: цепь, цикл, звезда, полный граф (синтетические топологии) и бенчмарке IMDB (реальный граф отношений), чтобы выявить влияние топологии на качество/время.

3. Зафиксировать эмпирическое правило: для простых топологий (цепь и цикл) динамическое программирование доминирует по стоимости и часто по времени планирования, а для более сложных (звезда, полный граф) динамическое программирование остаётся лучшим по стоимости, но начинает резко проигрывать по времени планирования при росте числа отношений.

Преимущества жадного подхода

1. Полиномиальное время планирования и хорошая масштабируемость по времени планирования на больших запросах.

2. Топологии показывают, что порог по числу таблиц недостаточен — важна форма графа соединений.

Недостатки

1. Качество планов по стоимости хуже, чем у DP почти на всех топологиях.

2. На IMDB жадный подход нестабилен по стоимости. Автор фиксирует большую разницу стоимости относительно синтетических топологий и существенно более высокие стоимости у GOO на большинстве размеров.

Адаптивная оптимизация очень больших запросов соединения

В статье "Adaptive Optimization of Very Large Join Queries" — Neumann & Radke (2018) предлагается адаптивный фреймворк оптимизации порядка соединений, который выбирает (и переключает) стратегию планирования по сложности графа соединений и заданному бюджету перебора, чтобы для типичных запросов получать оптимум, а для больших деградировать по качеству плавно и предсказуемо по времени оптимизации.

Понятие адаптивности

Под адаптивностью в работе понимается не фиксированное правило вида "DP до N таблиц, дальше эвристика", а пер-запросный выбор алгоритма по структуре (топологии) графа запроса и контроль затрат оптимизации через бюджет перечисления. В результате "малые/простые" запросы решаются точно, а "длинный хвост" (сотни и тысячи отношений) обрабатывается эвристически, но с контролируемым временем оптимизации.

Ключевые идеи и особенности адаптивного планирования

1. Оценка сложности через число связных подграфов и бюджет 10 000.
   Авторы считают (с ранней остановкой) #ccp графа соединений. Это число совпадает с размером полной DP-таблицы (а значит, с памятью DP и косвенно со временем оптимизации). Если число связных подграфов не превышает 10 000, то графовый DP считается "достаточно быстрым", и запрос оптимизируется точно.

2. Адаптивная стратегия (decision tree) вместо порога по числу отношений.
   Для $|V| < 14$ DPHyp запускается без подсчёта (клика — худший случай, и примерно до 14 отношений DP ещё реалистичен). Для запросов до 100 отношений используется подсчёт #ccp с бюджетом 10 000; при успехе — точный DPHyp, иначе происходит переход к следующей стадии. При наличии внешних соединений фреймворк выбирает другой путь.

3. Search space linearization для "medium" запросов.
   Когда полный DP становится слишком дорогим, вводится линеаризация пространства поиска: сначала строится линейный порядок отношений, затем DP ограничивается только связными подцепочками этого порядка (connected subchains), вместо произвольных подмножеств. Это уменьшает размер DP-таблицы с $O(2^n)$ до $O(n^2)$ (и делает время порядка $O(n^3)$). Качество плана зависит от выбранного порядка: при "плохой" линеаризации некоторые хорошие порядки соединений становятся недостижимыми. Кроме того, гипер-рёбра не выражаются линейно, поэтому linearized DP применима только к обычным графам соединений.

4. GOO + локальная оптимизация больших поддеревьев под бюджет (для "large/mega").
   Для очень больших запросов строится начальный план жадной эвристикой (GOO), после чего выполняется итеративное улучшение: выбранные (дорогие) поддеревья перепланируются более точным методом DP, но только до размера k. В обычном случае в роли "внутреннего DP" используется linearizedDP и берётся $k \approx 100$, что позволяет исправлять крупные фрагменты плана. Если в запросе есть не внутренние соединения, то вместо linearizedDP приходится использовать DPHyp, поэтому берут существенно меньший $k \approx 10$. Бюджет оптимизации расходуется явно: после каждого вызова внутреннего DP уменьшают оставшийся бюджет, что обеспечивает ограничение по времени оптимизации и "плавное" ухудшение качества без резкого обрыва.

Преимущества

1. Оптимальность для частого случая: если граф достаточно простой по #ccp бюджету, фреймворк гарантированно запускает DPHyp и находит оптимальный порядок.

2. Масштабирование до тысяч отношений: при росте размера запроса происходит переход к менее дорогим стадиям (linearizedDP, затем GOO+DP на поддеревьях), что позволяет работать с "длинным хвостом" размеров запросов.

3. Плавная деградация качества: вместо жёсткого переключения "DP $\rightarrow$ greedy" вводятся промежуточные стадии и улучшение крупных подпроблем DP, что снижает разрыв по качеству планов.

4. Учитывается форма графа запроса: выбор алгоритма определяется не только числом начальных отношений, что точнее отражает реальную сложность перечисления.

Ограничения и недостатки

1. Ограниченная применимость линеаризации: linearizedDP неприменима при внешних соединениях и гипер-рёбрах, поэтому для таких запросов качество/скорость хуже (внутренний DP тяжелее, а k приходится уменьшать).

2. Зависимость качества от линеаризации: выбранный линейный порядок ограничивает пространство планов, следовательно, некоторые хорошие порядки соединений становятся недостижимыми.

3. Локальность улучшений в GOO-DP: перепланирование поддеревьев размера k не даёт полной свободы глобальных перестановок между различными поддеревьями, поэтому итог может оставаться далёким от оптимума на "трудных" структурах.

4. Пороговые параметры являются эвристическими: бюджет, границы и k определяют поведение и требуют настройки под конкретную систему и модель стоимости.

Оценки в модели стоимости

[How Good Are Query Optimizers, Really?]

Оптимизатор СУБД выбирает план выполнения по стоимостной модели, опираясь на оценки селективностей и кардинальностей промежуточных результатов. Если эти оценки сильно ошибочны (что часто происходит из-за предположений о независимости предикатов и упрощённых статистик), оптимизатор может выбрать плохой порядок соединений и неподходящие алгоритмы, что приводит к кратному (иногда на порядки) замедлению выполнения запроса.

Авторы предлагают измерять качество оптимизатора не абстрактно, а на контролируемой постановке: берётся набор реальных сложных запросов и сравниваются (1) ошибки оценок кардинальностей и (2) итоговое время выполнения при разных условиях. Используется Join Order Benchmark (JOB) на базе IMDb: 113 запросов с множественными соединениями и фильтрами. Ключевой приём: модифицируется PostgreSQL так, чтобы оптимизатор мог получать *истинные* кардинальности (как будто

статистика идеальна). Это позволяет отделить влияние ошибок оценок от прочих факторов и ответить на вопрос: насколько улучшится план и время выполнения, если убрать только ошибки селективности/кардинальности. Дополнительно анализируется роль качества стоимостной модели и объёма перебора планов (полный DP-поиск против ограниченных эвристик).

Выводы:

1. Ошибки оценок кардинальностей встречаются часто и могут быть очень большими, особенно в запросах с множеством соединений и коррелированными условиями.
2. Главный источник плохих планов — именно ошибки кардинальностей: при доступе к точным размерам оптимизатор существенно чаще выбирает правильный порядок соединений и более подходящие алгоритмы, что заметно ускоряет выполнение.
3. Точность самой стоимостной функции оказывает меньший эффект: упрощение параметров стоимости обычно меньше влияет на качество плана, чем исправление кардинальностей.
4. Полноценный перебор (динамическое программирование для join order) даёт преимущество над ограниченными стратегиями поиска: даже при неточных оценках он реже пропускает хороший порядок соединений, тогда как ограниченный поиск может застрять на неудачном плане.
5. При этом оптимизатор часто остаётся "достаточно хорошим" на практике: многие планы устойчивы к ошибкам (например, хеш-соединения), а катастрофические провалы возникают в сравнительно редких сочетаниях (глубокие деревья соединений + сильные ошибки + рискованные выборы вроде больших nested-loop).